POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Message Flow Analysis for Distributed Real-Time Control Systems

CHRISTOPHE BOURQUE BÉDARD
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Juin 2022

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Message Flow Analysis for Distributed Real-Time Control Systems

présenté par **Christophe BOURQUE BÉDARD**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

DEDICATION

« #yopo »
- Eva T.

ACKNOWLEDGEMENTS

This work was made possible by many, many people. I first got introduced to the world of tracing and performance analysis through my internship with the Trace Compass team at Ericsson in 2018. In turn, they introduced me to the DORSAL lab and Michel Dagenais, whom I thank for allowing me to work on an undergraduate research project exploring tracing with ROS 1 back in 2018-2019, thus eventually leading to this project. I am also grateful to Ingo Lütkebohle for giving me the opportunity to apply tracing to ROS 2 at Bosch in Germany in 2019.

Furthermore, I would like to thank Michel Dagenais again for his mentorship and for giving me a lot of leeway all through this project. I would also like to acknowledge Ericsson, the Natural Sciences and Engineering Research Council of Canada (NSERC), and Prompt for the financial support.

All of this would not have happened had I not had the real pleasure of working on autonomous drones with Élikos, a team of passionate students at Polytechnique. The challenges, the people I met, and the friends I made were wonderful and led me to embark on this amazing journey.

I wish to thank my friends Pierre-Yves Lajoie and Eva Terriault for their kindness and support throughout this unusual (in every way) adventure of mine. Finally, I would like to thank my family for enduring my never-ending student status, and S. for always sticking with me through thick and thin.

RÉSUMÉ

Au courant des deux dernières décennies, le développement logiciel en robotique, effectué par des chercheurs et des compagnies à travers le monde, a évolué, en commençant par des solutions sur mesure pour se diriger vers des cadres d'applications haut niveau libres. Ceci est devenu particulièrement utile pendant la dernière décennie, puisque les applications robotiques sont devenues de plus en plus complexes, en partie grâce à des avancées technologiques au niveau des capteurs, actionneurs, et des plateformes de calcul. Malheureusement, les cadres d'applications tels le *Robot Operating System (ROS)*, qui utilisent les mécanismes *publisher-subscriber* et *remote procedure call (RPC)*, présentent plusieurs défis au niveau de la performance. En effet, ils effectuent l'ordonnancement haut niveau par-dessus l'ordonnanceur du système d'exploitation, ce qui peut mener à des inefficacités et des goulots d'étranglement au niveau de la performance. Les outils usuels de débogage fournissent une vision étroite de l'exécution de l'application; ils ne peuvent pas procurer une vue d'ensemble de l'exécution d'une application distribuée. Les techniques et outils de débogage et d'analyse de performance doivent donc être adaptés pour ce genre de cadre d'application. De plus, ces outils ne sont souvent pas compatibles avec les contraintes de systèmes temps réel : ils doivent avoir un surcoût d'exécution minime afin d'éviter d'affecter l'exécution d'une application et de fournir des résultats valides. Les méthodes existantes pour ROS 2 se concentrent sur des cas d'utilisation très spécifiques, ce qui les rend inadéquates pour l'analyse de performance générale, en plus d'avoir un grand surcoût d'exécution.

Afin d'améliorer la littérature sur ce sujet, ce mémoire apporte deux contributions principales. Premièrement, il introduit un nouvel ensemble d'outils de traçage pour ROS 2. L'instrumentation polyvalente pour traçage proposée permet d'extraire de l'information sur l'exécution de ROS 2, et des expérimentations démontrent son bas surcoût d'exécution. De plus, des outils de traçage permettent de configurer le traçage à travers le système d'orchestration puissant de ROS 2, ce qui est primordial pour une utilisation efficace par des chercheurs et d'autres utilisateurs. Ensuite, la deuxième contribution est une analyse du flot de messages à travers un système distribué ROS 2. Elle permet d'extraire de l'information haut niveau sur l'exécution, ce qui aide à identifier des causes potentielles de problèmes de performance, en plus de permettre d'étudier l'ordonnanceur haut niveau de ROS 2. Elle inclut aussi un modèle abstrait de l'exécution d'une application ROS 2, qui peut être utilisée pour d'autres analyses, et inclut une expérimentation qui démontre encore une fois le faible surcoût d'utilisation de la méthode. D'autres expérimentations sur des systèmes robotiques synthétiques et réels démontrent son potentiel dans un but d'optimisation de performance et, en général, dans le but de comprendre l'exécution d'un système ROS 2. Ces contributions pourraient donc être exploitées par d'autres chercheurs et développeurs afin d'étudier et d'améliorer la performance de ROS 2 et d'autres cadres d'application similaires.

ABSTRACT

Over the last two decades, software development in robotics has shifted from a focus on building custom solutions – thus reinventing the wheel frequently – to building open-source high-level modular frameworks that are used and improved by researchers and companies all over the world. This has become particularly useful during the last decade, since robotics applications have gotten significantly more complex, in part due to technological advances with sensors, actuators, and computing plaftorms. Unfortunately, robotics software frameworks like the Robot Operating System (ROS), built on the publisher-subscriber and remote procedure call (RPC) paradigms, have multiple performance challenges. Indeed, they perform high-level scheduling on top of the operating system scheduler, which can lead to major performance inefficiencies and bottlenecks. Common debugging tools provide a very narrow view of the application execution; they cannot provide a global perspective on the execution of a distributed system. Software debugging and performance analysis tools must be adapted for such frameworks and evolve alongside them. Furthermore, performance analysis tools are often not compatible with strict constraints of real-time – and potentially safety-critical – applications: they must have minimal runtime overhead to avoid perturbing the application and to provide valid results. Current methods for ROS 2 focus on very specific use-cases, thus making them unsuitable for general performance analysis, and generally have high runtime overheads.

To improve the literature on this subject, this thesis brings two main contributions. It first introduces a novel multipurpose tracing framework for ROS 2. The proposed multipurpose tracing instrumentation allows extracing ROS 2-level execution information, and experiments demonstrate its low overhead. Furthermore, tracing tools allow configuring tracing through the powerful ROS 2 orchestration system, which is indispensable for effective use by researchers and other end-users. Then, the second contribution is a message flow analysis, showing the path of messages across distributed ROS 2 systems. It extracts high-level information, which helps identify potential causes of performance bottlenecks, and can be used to study the ROS 2 scheduler. It also includes an abstract model of the execution of a ROS 2 application, which can be used for other analysis goals, and includes an experiment to again demonstrate the low runtime overhead of this method. Further experiments on both synthetic and real robotic systems demonstrate its potential for performance optimization, and in general for understanding the execution of a ROS 2 system. These contributions could therefore be leveraged by other robotics researchers and software developers to study and improve the performance of ROS 2 as well as other similar frameworks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

API	application programming interface
AUTOSAR	AUTomotive Open System ARchitecture
CORBA	Common Object Request Broker Architecture
CTF	Common Trace Format
DAG	directed acyclic graph
DDS	Data Distribution Service
DLT	Diagnostic Log and Trace
IDL	interface definition language
IoT	Internet of things
IPC	inter-process communication
LET	logical execution time
LTTng	*Linux Trace Toolkit: next generation*
MPI	Message Passing Interface
OMG	Object Management Group
OS	operating system
pub-sub	publisher-subscriber
QoS	quality of service
RCU	Read-Copy-Update
ROS	Robot Operating System
RPC	remote procedure call
RTOS	real-time operating system
RTPS	Real-Time Publish Subscribe
SLAM	simultaneous localization and mapping
SOME/IP	Scalable service-Oriented MiddlewarE over IP

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

The last two decades have brought significant progress in the field of robotics. Lower costs and newer & better technologies have transformed robotics into an ever-expanding industry. On the development side, there has been a shift from the build-everything-from-scratch mentality to using higher-level open-source frameworks. The wheel is thus no longer reinvented by everyone, which cuts down on implementation time. This helps both academic and industrial research as well as product development.

Alongside those technological advances – and in part as a result of them – robotics applications have also evolved and gotten more complex. Modern applications have a much larger scope, are more computationally-intensive, and might involve distributed or multi-robot systems. This includes domestic tasks, agriculture, manufacturing, warehouse automation, search & rescue, and even space exploration. It also includes safety-critical applications such as autonomous driving and uncrewed aerial systems. Software debugging and performance analysis tools & techniques must adapt and evolve as well.

This thesis includes two research articles (Chapters 4 and 5):

1. Christophe Bédard, Ingo Lütkebohle, and Michel Dagenais, "ros2_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6511–6518, 2022.
2. Christophe Bédard, Pierre-Yves Lajoie, Giovanni Beltrame, and Michel Dagenais, "Message Flow Analysis with Complex Causal Links for Distributed ROS 2 Systems," *Robotics and Autonomous Systems*, [in review] 2022.

The research work presented in this thesis was done in the DORSAL laboratory at Polytechnique Montréal (Montréal, Québec, Canada) under the supervision of Michel Dagenais. Aside from our great industrial partners and research lab colleagues, this work also benefited from collaborations with Ingo Lütkebohle from Bosch Corporate Research and Pierre-Yves Lajoie from MIST lab (Polytechnique Montréal).

1.1 Definitions and Basic Concepts

This section goes over basic concepts and definitions that are useful for understanding the problem statement as well as the rest of this thesis.

1.1.1 Robotics

Typical robotic systems consist of three distinct elements: sensors, processors, and actuators. Sensors convert observations of the real world into data. This data is then transformed into commands using any number of processing layers. Commands are sent to actuators, which physically move the system, thus affecting future sensor readings. Robotic systems can have multiple sensors and actuators, and processing can be quite complex. For example, simultaneous localization and mapping (SLAM) is used to map an unknown environment using sensor data and then localize the robot within that map. Subsequent processing layers then use this high-level localization information and eventually produce actuator commands to move within the environment. Complex SLAM systems can be used to do surveying and infrastructure inspection [2], and can involve multiple robots all working on the same map [3].

1.1.2 Middleware

Middlewares[1] are simply an intermediary between two or more software or hardware components. They help abstract away some of the work to allow easily building on top of them, e.g., by providing an application programming interface (API).

Communications

In the field of robotics and Internet of things (IoT), middlewares are used to exchange messages between nodes, with the exact definition and mapping between nodes and machines, processes, or threads being implementation- or application-dependent. The two main methods are publisher-subscriber (pub-sub) and remote procedure call (RPC). With the pub-sub paradigm, messages are sent unidirectionally between one node and N nodes on a given topic, as shown in Fig. 1.1. Nodes can have multiple publishers and multiple subscribers, and any number of publishers or subscribers can publish or subscribe to a topic.

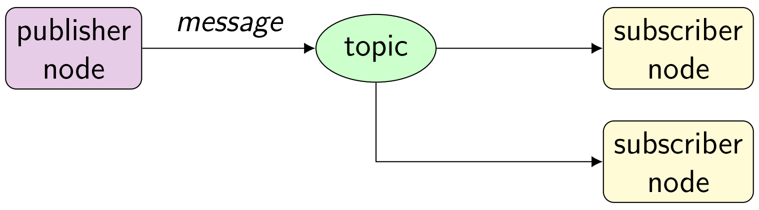


Figure 1.1 Typical publisher-subscriber communication.

[1]Henceforth intentionally pluralized with an s to refer to different middleware types and implementations.

On the other hand, as shown in Fig. 1.2, RPC allows bidirectional, request-reply communications, for example to request data from – or delegate computation to – another node. It is therefore an additional abstraction on top of the pub-sub paradigm, providing a request topic and a reply topic.

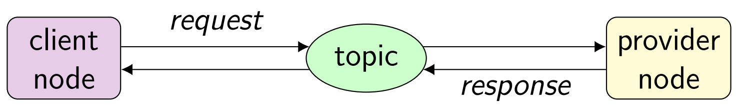


Figure 1.2 Typical RPC communication, with a request and a reply.

Orchestration

Message-based communications allow building modular systems, which aids development. Orchestration systems are often used to assemble these multi-node systems. This can be especially convenient when multiple hosts are involved.

1.1.3 Distributed Systems

In distributed systems, components of a larger system are spread across multiple computers over a network. This can be used for many types of applications, including robotics. In simple applications, this can be achieved by having multiple computers (not necessarily a coprocessor) in a single robot that communicate over a network switch. In more complex applications, these are multi-robot systems working on a single common task communicating over a network.

1.1.4 Real-Time Systems

In the world of embedded and safety-critical systems, real-time systems have very stringent requirements. Real-time systems must have response time guarantees, and usually have short and strict deadlines. For example, for an autonomous driving system, if an obstacle is detected, the vehicle must react within a very short amount of time. Part of what is needed to accomplish this is determinism: for a given input, the system must always provide the same output. This includes not having dynamic memory allocations or blocking calls during runtime. The former are non-deterministic and can fail; dynamic memory allocations are restricted to the initialization phase, or applications are simply designed to only use static memory. The latter may of course block for any amount of time; reasonable timeouts can be used, or blocking calls can be replaced with polling calls. This thus applies at all levels, from the drivers and operating system up to the application itself.

1.1.5 Software Debugging and Tracing

There are many software debugging techniques. Common tools such as debuggers give control of the execution to developers in order to introspect and understand the execution of specific parts of the code. Observability tools provide information about the execution of an application. This can be achieved by sampling the state of an application periodically, by using counters to record basic information, or by recording events.

Tracing is a form of fast low-level logging. Applications are instrumented in specific locations. Instrumentation can be static (i.e., included in the source code) or dynamic (i.e., added to the application during execution). Both userspace applications and operating systems can be instrumented and traced. When instrumentation points are executed, an event is generated and recorded. Each event contains a basic payload (e.g., timestamp, event type or name) and a payload that is specific to the given instrumentation point. Traces are collections of these events. While logging allows recording user-readable information (i.e., strings), tracing usually records raw data.

1.1.6 Trace Analysis

Analyzing data from the trace helps provide helpful information. It can be done online, in which case it is considered monitoring. However, in most cases, it is done offline, i.e., after the fact, since it is simpler and leads to less runtime overhead.

Traces can be analyzed to create a model of the execution. This can produce statistics about parts of the application, and can also produce higher-level visualizations of the behaviour of an application.

1.2 Problem Statement

1.2.1 High-Level Task Scheduling

Modern robotics applications run on more common kernels like Linux and operating systems like Ubuntu instead of baremetal microcontrollers. Operating system scheduling, combined with the high-level scheduling of tasks in publisher-subscriber frameworks, is challenging. There are multiple engineering challenges and open research problems related to this. There is therefore a need for improved tools and analysis methods to study this.

1.2.2 Extraction of Execution Information

Extracting execution information from an application must have minimal runtime overhead to avoid perturbing the application, especially in safety-critical applications. The observability tool must also be compatible with real-time principles. This includes both the choice of tracer and the design of the instrumentation.

Furthermore, robotics software frameworks have a complex architecture. Multiple abstraction layers make efficient instrumentation more challenging, requiring that multiple locations in the source code be instrumented.

1.2.3 Tracing Configuration and Tooling

The modularity of middleware-based robotic systems requires powerful but flexible tools. Orchestration systems make putting together and executing large modular systems easy, either on a single host or on multiple hosts. Common tools to configure tracing are created to be universal, hence they are unfortunately simple and rudimentary. They need to be adapted to the aforementioned use-cases.

1.2.4 Trace Analysis

Tracing generates considerable amounts of data. This data needs to be processed and analyzed in order to provide useful and actionable information. The research work presented in this thesis aims to provide high-level information on the execution of a ROS 2 application. This includes information on the path of a message across a ROS 2 system, which could be distributed. To be able to dig deeper, it should be possible to correlate such information with execution information from the operating system.

1.3 Research Objectives

This thesis, including articles in Chapters 4 and 5, aims to tackle the following research objectives:

- Develop low-overhead instrumentation that allows extracting execution information from ROS 2.
- Develop or adapt tracing tools to conform to the intricacies of typical distributed robotic systems.
- Develop an abstract model of the execution of a ROS 2 application.

- Develop an algorithm to extract high-level information from the abstract model: path of message across a distributed robotic system.
- Perform experiments on both simulated and real robotic systems for validation.

1.4 Thesis Outline

This thesis contains 6 other chapters. Chapter 2 provides a review of the literature, showing the state of the art in performance analysis and robotics. Then, a summary of the research approach is presented in Chapter 3. Following this approach, Chapter 4 and Chapter 5 each present a research article. Chapter 6 contains a general discussion on the results of the two papers. Finally, Chapter 7 concludes and mentions limitations as well as possible future work.

CHAPTER 2 LITERATURE REVIEW

2.1 Tracing and Profiling

Extracting performance-related execution information can be done using tracing and profiling, as introduced in Section 1.1.5. However, it must be minimally invasive to avoid perturbing or influencing the system when observing it. Otherwise, the system may not work as expected and the results may be invalid. This is known as the observer effect [4, Section 2.3.10] and can manifest as "heisenbugs" [5, Section 11.6.4], although the latter inaccurately refers to Heisenberg's uncertainty principle [6].

2.1.1 Profiling

Profiling is a great general-use observability technique. It provides execution information by collecting samples from various sources [4, Section 4.2.2]. With sampling-based profiling, samples are usually collected at a fixed rate. Unfortunately, if the rate of the target activity is similar to the sampling rate, sampling can completely miss it, leading to erroneous results. This is somewhat analogous to the Nyquist–Shannon sampling theorem in the field of signal processing [7].

CPU profiling is a common form of profiling. It can provide a breakdown of CPU usage by function. `perf` [8] and `gprof` [9] are common profilers. The information can be visualized using flame charts and flame graphs [10, 11]. The former show the state of the call stack frame over time, while the latter show the total time of each function in the call stack.

Profilers can also provide information about performance counters, e.g., for cache. However, this is intended for lower-level performance tuning.

2.1.2 Tracing

Tracing is event-based, collecting events using instrumentation instead of sampling data [4, Section 4.2.3]. It can be done at different levels, e.g., kernel and userspace. Instrumentation can be static (i.e., in the source code, called tracepoints) or dynamic (i.e., added during runtime). Static instrumentation can also be integrated using the `LD_PRELOAD` mechanism, for example to intercept certain function calls, trigger tracepoints, and then call the original function. In-source static instrumentation is common when analyzing large applications as a whole.

Tracing can also be used for function profiling by instrumenting function entries and exits, e.g., using the `-finstrument-functions` `gcc` option. Flame charts and flame graphs can be produced from this information as well. While this provides the real number of calls, such instrumentation also adds significant runtime overhead. Since the overhead cost is the same for each instrumented function, this skews function duration results, especially with high numbers of short function calls. Tools exist to selectively instrument functions instead of instrumenting all functions [12], but this still must be kept in mind when interpreting the results.

There are many tracers, each one with a unique set of features and underlying mechanism [13]: `perf` [8], DTrace [14], Ftrace [15–18], `strace` [19–21], VampirTrace [22], SystemTap [23–25], eBPF [26, 27], and *Linux Trace Toolkit: next generation* (LTTng) [28–30]. `perf` and DTrace can trace various events (on top of the already-mentioned profiling feature). Ftrace traces kernel functions, while `strace` traces system calls. VampirTrace was created to trace large-scale applications leveraging low-level parallel programming tools such as the Message Passing Interface (MPI) [31]. SystemTap and eBPF allow tracing various events. eBPF is an *eierlegende Wollmilchsau*[1] kind of tool, meaning that it can do anything. However, while it is useful for general-purpose system performance analysis, at its core, eBPF is an aggregator for live monitoring. For instrumenting & tracing applications, other tools are more fitting.

LTTng

LTTng [28–30] is a low-overhead tracer [13] with a design that is compatible with real-time requirements [32]. It is reetrant [33], thread-safe, signal-safe, non-blocking, has no system calls in the fast path, and does not make copies of the trace data. It uses Read-Copy-Update (RCU) [34] for concurrent wait-free reads of internal variables [35]. LTTng writes trace data to ring buffers, which are then usually consumed by being written to the file system or sent over the network to be consumed by another system. It uses a tracepoint mechanism for instrumentation, which is the most efficient mechanism according to Gebai and Dagenais [13]. Similar to other tracers, tracepoints can be enabled or disabled during runtime. When disabled, tracepoints have virtually no performance impact [13]. LTTng as a whole has both a kernel tracer and a userspace tracer, both generating Common Trace Format (CTF) [36] traces. The userspace tracer is completely implemented in the userspace, thus avoiding costly context switches. Developers can design instrumentation that is specifically tailored to the needs of their application and then implement it by adding static LTTng tracepoints. Furthermore, while LTTng is most commonly used for offline analysis, in

[1]German for “egg-laying wool-milk-sow,” literally.

which all trace data is recorded and written to disk before being processed, LTTng also has a snapshot mode. This mode is similar to how flight recorders work: trace data is written to a buffer, and the buffer gets overwritten with new data until a snapshot is requested. This can be particularly useful for production settings, when only the most recent chunk of data is required, and it also avoids writing the data to the file system when it is not needed.

Finally, kernel and userspace traces collected using LTTng can easily be analyzed together [37, 38]. The Babeltrace tool [39], especially Babeltrace 2, can be used to read and process CTF traces. However, there are better tools for more complex trace analyses.

2.2 Trace Analysis

As defined in Section 1.1.6, traces contain raw, low-level information, and must therefore be processed to produce useful information for users and developers. Traces are processed from beginning to end; the result of this is often a time-based chart showing events and states over time. Statistics can also be computed and shown in XY charts.

2.2.1 Trace Analysis Frameworks

As with tracers, there are many trace analysis tools and frameworks, each with a unique set of features and general approach. KernelShark [40] can analyze kernel traces obtained with `trace-cmd` [41], an Ftrace frontend. Vampir [22, 42–44] encompasses VampirTrace to form a framework for instrumenting, tracing, and analysis. Tracealyzer [45] is a trace viewer built for lower-level real-time operating systems (RTOSs). Diagnostic Log and Trace (DLT) [46] is an all-in-one tracing & loging and viewing framework built specifically by and for the automotive sector under AUTomotive Open System ARchitecture (AUTOSAR), an automotive consortium. These trace analysis frameworks are generally too specific in scope, and too limited in features and extendability.

Trace Compass

Trace Compass is a full framework for trace data analysis [1]. It can process many trace formats, including CTF traces. It has many built-in analyses and views, including a *resources view*, showing the state of each CPU over time, and a *control flow view*, showing the state of threads over time. Most of the built-in views show an abstract version of the execution trace with time-based data in the form of segments, similar to a Gantt chart. However, it can also generate tables and XY charts.

Furthermore, users can extend Trace Compass by providing their own analysis and viewer plugins. Analyses can depend on any number of other analyses: users can use the result of those analyses as an input for their own processing, thus favouring re-use and modularity. Trace Compass offers many tools and APIs that are commonly needed for trace data processing. It provides a straightforward way to process a trace by reading events one by one, in order. This allows collecting statistics or building a model using both simple and complex state machines. It also provides a mechanism for storing time-based data in a tree structure on disk or in memory. Custom analyses can therefore easily leverage existing analyses and tools, which reduces development time.

Finally, Trace Compass is being integrated as a plugin for the Eclipse Theia cloud IDE platform [47]. Decoupling the trace viewer frontend from the trace analysis backend has many benefits. For example, it enables large-scale distributed processing of very large traces. This also follows a general workflow change in the industry, where companies are shifting from managing individual installations of tools to providing centralized tools with access to more substantial computing resources.

2.2.2 Trace Analysis Methods

In general, traces can be either processed directly or processed to create an abstract model. However, there are a number of issues and limitations with this approach. Direct analyses are limited by what was observed, and model-based analyses are limited by the conditions under which the trace was made [48].

Studying distributed applications (see Section 1.1.3) as a whole can provide very useful information. To do so, each individual system must be traced, and then the traces can be combined and analyzed [4, Section 5.4.8]. Since each system has its own clock, traces have to be synchronized in order to be correctly correlated. Trace synchronization can be performed using the convex hull algorithm using network packets [49], both offline [50] and online [51]. This can be applied to heterogeneous embedded systems [52].

Furthermore, real-time applications were traced and studied in [53–57]. Finally, combining execution information from kernel traces and userspace traces can help solve some of the aforementioned trace processing issues [58].

Critical Path

The critical path is defined as the longest path in a directed acyclic graph (DAG). Therefore, if this path is shortened, the overall length from beginning (i.e., root) to end is shortened.

The initial definition and application comes from task scheduling for project activities in management [59]. A similar idea was applied to circuits by Abramovici *et al.* [60] to identify faults detected during tests. Of course, the method can be applied to software [61], where identifying the critical path is helpful for optimization. Yang and Barton [62] applied it to compute the critical path of the execution of parallel and distributed programs. They built a DAG of the execution history of an application using data collected during execution. Graph edges represent an activity with a certain duration, either executing or waiting. Graph vertices represent activity beginnings or ends: program start or end, or network send or receive. The longest *execution* path can then be computed to identify the bottleneck.

Hollingsworth [63] performed an online computation of the critical path for a message-passing parallel program using information obtained from logs. Saidi *et al.* [64] improved the technique, targetting systems with numerous interacting state machines. Fournier and Dagenais [65] applied a similar method to identify and analyze blocking in parallel programs using kernel traces obtained with LTTng. Giraldeau and Dagenais [66] extended this method and used wait-related events from the kernel (e.g., scheduling, network, or interrupts) to recursively compute wait dependencies across machines, thus identifying the critical path. One of the main advantages of this kernel-only approach is a userspace-independence, since all userspace applications use these operating system primitives. Therefore, userspace applications do not need to be instrumented. Doray and Dagenais [67] further extended this method to identify differences between multiple executions of the same task using both kernel and userspace traces. Nemati *et al.* [68] applied the algorithm to virtualized environments using kernel tracing. Ezzati-Jivan *et al.* [69] proposed a method to model the wait dependencies between threads and hardware resources using kernel traces.

2.3 Middleware

As defined in Section 1.1.2, middlewares enable message-based communications between components. There are two main types: centralized and decentralized.

2.3.1 Centralized Middleware

The first middleware type is broker-based, where a central server coordinates communications between components. MQTT [70,71] and other variants offer pub-sub messaging, usually over TCP. This lightweight and efficient protocol is designed for low-level, low-energy IoT devices running on potentially unreliable networks.

2.3.2 Decentralized Middleware

On the other hand, decentralized middlewares do not use central message brokers, which of course comes with additional complexity. LCM [72, 73] provides pub-sub communications and is designed to be integrated into any kind of application. Scalable service-Oriented MiddlewarE over IP (SOME/IP) [74], another AUTOSAR standard, was built for automative applications, and provides pub-sub and RPC communications over TCP or UDP.

DDS

Data Distribution Service (DDS) [75,76], an Object Management Group (OMG) standard, is designed for flexibility and scalability, and uses a discovery mechanism to achieve complete decentralization. While DDS is used for IoT applications, it was intended for more powerful applications and does not exclusively target unreliable networks. DDS itself incorporates many other OMG standards, including a topic-level security specification, an interface or message definition language borrowed from the Common Object Request Broker Architecture (CORBA) [77] interface definition language (IDL), and Real-Time Publish Subscribe (RTPS) for the actual message transport. DDS supports TCP and UDP, and can leverage both unicast and multicast, which is key for scalability with large numbers of nodes. It provides its own quality of service (QoS) policies and offers a much wider range of policies compared to SOME/IP, which only proposes a reliability setting. DDS offers QoS policies related to reliability, deadlines, message durability, message history depth, data lifespan, liveliness, etc. There are multiple implementations of the DDS standard: Eclipse Cyclone DDS [78], eProsima Fast DDS [79], and RTI Connext DDS [80], to name a few. The first two are open-source, while the last one is proprietary. Moreover, the RTPS protocol was created to be interoperable, but the full features of different DDS implementations might not be.

2.4 Robotics

A number of frameworks and tools have been created to help ease software development in robotics. Player, a robotics framework, and Stage, a simulator, commonly referred to as Player/Stage [81], were first created in 1999. Their main goal was to provide a common open-source infrastructure for multi-robot systems research. However, since Stage is a 2D simulator for indoor environments, its applications are limited.

2.4.1 Robot Operating System

Robot Operating System (ROS) [82] is an open-source robotics framework that was first released in 2007. It is a middleware and set of tools for robotics software development. It offers pub-sub and RPC-like communications between nodes (see Section 1.1.2), using publishers, subscriptions[2], services, and actions. Communication topics have a name and a message type, e.g., an integer, a string, or a complex structure defined using a simplified message description language similar to the CORBA IDL. Services and actions are both asynchronous request-reply RPCs, with the latter having optional progress feedback messages. Service servers offer a unique service which can be used by service clients. As mentioned in Section 1.1.2, all of this enables the creation of high-level computation graphs, facilitating modularity. Furthermore, since ROS is open-source and has garnered many users over the years, implementations for common algorithms and sensor drivers can be easily found and used. This open-source ecosystem thus significantly promotes reusability and is very attractive to potential users [83].

ROS was created as an academic research tool, and was not designed for real-time applications (see Section 1.1.4). Its custom message-passing middleware implementation does not offer the features, flexibility, or performance that are commonly offered by state-of-the-art middlewares. Moreover, it uses a central broker for discovery of pub-sub and RPC objects; each machine needs to be configured to point to that central node. Although this is acceptable for systems consisting of a few machines, it considerably limits scalability. Consequently, ROS did not meet the requirements of real-time safety-critical applications.

Therefore, in 2014, development began for ROS 2 [84, 85] as a complete re-write of ROS (i.e., ROS 1) to meet the requirements of those newer use-cases and to leverage new technologies [86]. The ROS 2 architecture is therefore substantially different [87]. While ROS 1 uses a custom message transport protocol with a central broker, ROS 2 defines a middleware interface and leaves the message-passing tasks up to an actual middleware [88]. The default middleware is DDS; a few implementations are available and tested for ROS 2: Cyclone DDS [78], Fast DDS [79], and Connext DDS [80]. Furthermore, ROS 2 defines its own IDL, although it is very similar to the message description language from ROS 1, and the mapping between the DDS IDL and the ROS 2 IDL is trivial. ROS 2 also defines and proposes its own QoS settings; in practice, this is a subset of the DDS QoS settings. However, since the middleware abstraction allows ROS 2 to run using any middleware, other non-DDS middlewares can also be used. For example, the Eclipse iceoryx [89] inter-process shared memory middleware can be used through rmw_iceoryx [90], an implementation of the middleware interface for iceoryx. Similarly, rmw_email [91, 92] leverages a middleware that uses emails

[2]ROS uses “subscription” instead of “subscriber” by convention.

to exchange messages.

ROS 2 also features “lifecycle nodes” [93], which are stateful nodes based on a standard state machine. Transitions between creation, configuration, activation, deactivation, shutdown, and error states can be triggered using services and other tools. This makes their life cycle well defined and easier to control, fitting with common approaches for safety-critical applications [94].

As introduced in Section 1.1.2, orchestration systems are used to assemble multi-node systems. Using a common system description language backend, the ROS 2 orchestration system, launch [95], supports description formats in XML and YAML files, i.e., launch files. This promotes aggregation and re-use of multiple smaller systems in a single host or as part of a larger distributed system. On the security side, ROS 2 exposes some of the DDS security features, namely authentication and authorization, through a user-facing tool, SROS2 [96].

Moreover, ROS 2 nodes may have any number of publishers, subscriptions, services, actions, or timer-triggered periodic callbacks. Nodes usually communicate with each other using messages. This means that the mapping between nodes and processes is not strict: users can define any number of nodes in a single executable. Nodes can also be made composable, making “components” [87], each one in its own shared library. Composition of nodes can then be done at runtime using command-line tools or defined in a launch file. Combining two components into a single process enables the use of intra-process communications, which can significantly improve latencies while still keeping the message-based abstraction.

Finally, while the underlying middleware takes care of sending and receiving messages, ROS 2 must coordinate the reception of messages internally. Messages and requests are passed on directly to the middleware, although they may actually be sent asynchronously. However, ROS 2 has to fetch new messages from the middleware: this is handled by the ROS 2 executor [97], a high-level scheduler. The default executor implementation is single-threaded, although a multi-threaded executor is also available. It constantly queries the middleware for new messages and checks if timers are due. Then, if a new message is available, it takes it from the middleware and calls the callback function of the corresponding subscription. Similarly, it calls the callback function associated with a timer if it is due. Furthermore, a static version of the single-threaded executor is also available. By not allowing nodes to be added to it after initialization, its internal logic can be simplified, reducing its runtime overhead and memory footprint compared to the default executor. Fortunately, as mentioned in Section 1.1.4, this fits with real-time principles, which can therefore easily take advantage of it. Consequently, the executor is a crucial part of the ROS 2 architecture and has a critical role in its performance.

2.4.2 ROS Systems

Multiple architectures for complete ROS 1- and ROS 2-based systems have been proposed. Kato *et al.* presented Autoware.AI [98, 99], an autonomous driving system completely based on ROS 1. Its successor, Autoware.Auto, is built on ROS 2. Similarly, a reference system based on the Autoware.Auto computation graph was proposed as a standard system for comparing and benchmarking different approaches [100]. Reke *et al.* [101] proposed an architecture based on ROS 2 with some real-time requirements. Finally, ROS 2 is also being used by NASA for space missions [102, 103].

2.4.3 ROS Performance

Network and Message-Passing

First, the networking layer was evaluated for real-time communications on Linux. Gutiérrez *et al.* [104,105] evaluated communications on a real-time Linux kernel using the PREEMPT_RT patch [106] and concluded that it can meet real-time requirements.

For ROS 2, Apex.AI proposed performance_test [107], a benchmarking tool to evaluate single pub-sub latencies over the network. Inspired by this, iRobot proposed a framework for evaluating latencies in custom computation graphs [108]. As will become evident in the following sections, the two approaches are different but complementary.

Maruyama *et al.* [109], Gutiérrez *et al.* [110], and Puck *et al.* [111] all evaluated the performance of ROS 2 as a whole and found it promising. Kim *et al.* [112] and Fernandez *et al.* [113] evaluated the performance impact of the ROS 2 security features. Thulasiraman *et al.* [114] evaluated the performance of ROS 2 and influence of its QoS settings on lossy networks. Barut *et al.* [115] compared the real-time capabilities of ROS 2 with PREEMPT_RT and OROCOS [116], a lower-level real-time robot control framework, and found that their performance was similar under normal conditions. Wang *et al.* [117] proposed a single-host inter-process communication (IPC) layer for ROS 1 and ROS 2 which reduces the overhead of IPC for large messages. Jiang *et al.* [118] proposed a serialization technique to improve communication performance by up to 93%. Kronauer *et al.* [119] evaluated the overhead of ROS 2 with relation to the underlying DDS implementation using profiling and showed that it can lead to 50% latency overhead. They noted that the message latency decreases as the publication frequency increases, potentially as a result of internal middleware buffers. They also noted that common operating system (OS) energy-saving features such as CPU frequency scaling cause a lot of latency variability. Finally, Puck *et al.* [120] noted in another performance evaluation of ROS 2 that the use of dynamic memory allocations accounts for a significant portion of the message processing time.

Executor

As introduced in Section 1.2.1 and Section 2.4.1, high-level scheduling of tasks with ROS is an open problem.

Exchanges of messages from node to node can be modelled as event chains and pipelines in DAG, which have been studied from a pure scheduling and dataflow perspective in the past [121–124]. Peeck *et al.* [125] focused on online monitoring for reacting to latency violations in event chains, proposing a monitoring solution that can identify deadline violations in order to try to recover. Casini *et al.* [126] proposed a formal scheduling model and a response-time analysis for ROS 2 to bound worst-case response times. Tang *et al.* [127] then proposed a more specific version that is however only valid for independent linear processing chains. Blass *et al.* [128] built on the work by Casini *et al.* [126] and proposed ROS-Llama, an online automatic latency manager for ROS 2. It extracts a model of the running system using tracing, and schedules threads to satisfy latency goals on a real-time system. Their work helped illustrate how the higher-level scheduling of tasks in ROS 2 does not apply very well to classic OS-level scheduling techniques. Blass *et al.* [129] further extended this work, and stressed how the ROS 2 executor differs from normal schedulers in the literature, since it prioritizes, in order: timers, subscriptions, service servers, and service clients. This inherent prioritization certainly impacts scheduling.

Lienen and Platzner [130] presented an FPGA-accelerated executor for ROS 2. Yang and Azumi [131] explored the real-time performance of the callback-group-level executor [132] available as an alternative in ROS 2, which allows having multiple distinct executor instances without interference. This allows setting scheduling priorities individually for each thread and its corresponding callback group, instead of bundling everything together as the default executor does. Choi *et al.* [133] proposed PiCAS, a priority-driven chain-aware scheduler, and showed that it helps lower end-to-end latencies. Staschulat *et al.* [134,135] proposed a budget-based executor for RTOSs based on the logical execution time (LET) paradigm [136, 137], which restricts communication of data to specific periodic time instances. Data is only read at the beginning of a time period and only written at the end. LET is more appropriate for programs relying on time or event triggers, thus helping make the executor more compatible with real-time OS schedulers.

Observability

Malavolta *et al.* [138] presented robotics software architecture guidelines that were mined from open-source projects. One of their guidelines mentions that logging should be standardized across a project and follow well-defined guidelines. As presented by Afzal *et al.* [139] and demonstrated by Quigley *et al.* [140], ROS users rely heavily on textual logging as well as tools such as rosbag, which records messages for later playback. Unfortunately, these methods provide execution information that is too abstract to be useful for performance-related concerns. Moreover, using the system itself to collect execution information – i.e., using publishers and subscriptions to record logs and messages – is also too invasive to be able to provide valid performance-related information, as explained in Section 2.1.

Blass *et al.* [128] noted that ROS-Llama improves latency-goal compliance, but that most of the runtime overhead of their solution is due to their custom tracer implementation. Forouher *et al.* [141] proposed a tool to visualize the flow of data in a ROS 1 system based on the data provenance principles from Acar *et al.* [142]. They do not consider the runtime impact, and their method relies on a special in-message header, which is invasive and does not fit with low-overhead observability principles. Similarly, Witte and Tichy [143] presented a tool to track messages in ROS 1 in order to interactively apply transformations. They also used a custom message header and noted that the performance overhead it introduces is significant.

Lütkebohle [144] identified a determinism problem with the standard obstacle avoidance algorithms in ROS 1. They used LTTng [28–30] to instrument ROS 1. They created an execution model and identified a lack of synchronization between input data and output data. This was presented as a generic tracing tool for ROS 1 [145], and was used and extended to visualize the flow of a message in ROS 1 [146].

Rivera *et al.* [147] proposed ROS-FM, which uses eBPF [26, 27] to monitor ROS-related network data for security purposes. Since they solely use network data, the applications of ROS-FM are limited. Furthermore, its overhead is at least 15% for ROS 2, which is significant for real-time systems. Nishimura *et al.* [148] proposed RAPLET, which provides a ROS-aware breakdown of the latency between publication and subscription in ROS 1. It uses `LD_PRELOAD` to dynamically instrument ROS and uses eBPF for network data. They claim that the pub-sub latency overhead of their tool is about 0.03 ms, which accounts for 2-20% of the latency depending on message size. They note that further enhancements are needed to enable the use of the tool in production.

2.5 Literature Review Summary

The previous sections provided an overview of multiple areas of the literature relating to the problem statement as presented in Section 1.2. LTTng has been identified as the most suitable tracer for instrumenting userspace applications and being able to correlate the information with other sources such as the Linux kernel. Trace Compass was also selected as the most powerful trace analysis framework. Moreover, various trace analysis methods were presented. Powerful methods use trace data to build an abstract model of the execution, which can then be used to provide useful information. The critical path method helps highlight the most important segments of the execution of a distributed application.

Thereafter, multiple middleware types were presented, and decentralized middlewares were established as being most appropriate for scalable or distributed systems. For this range of use-cases, ROS 2 on DDS was identified as the definite framework for robotics applications. Finally, open problems with ROS 1 and ROS 2 performance were presented. These problems predominently revolve around the handling of messages by the underlying middleware and the high-level scheduling of the ROS 2 executor. Some benchmarking and monitoring tools were proposed. However, there is no method that can produce a visualization of the flow of messages across distributed ROS 2 systems with a low performance impact.

CHAPTER 3 RESEARCH APPROACH AND THESIS ORGANIZATION

This chapter outlines the research approach used for the work presented in this thesis. It aims to explain the following two chapters, which correspond to the articles in Chapters 4 and 5, as well as the link between them.

3.1 Work Done

To accomplish the research objectives presented in Section 1.3, the approach and work for this thesis is split into two parts. In the first part, instrumentation and tools are created to collect execution data from ROS 2. The second part is about processing and analyzing the collected data. While these two main steps are not completely sequential, they are fairly distinct.

Following the literature review in Chapter 2, ROS 2 on Ubuntu, LTTng, and Trace Compass are chosen for the implementation. However, the approach would be applicable to other similar robotics frameworks, tracers, and trace analysis frameworks.

3.1.1 Instrumentation and Tools

First, the latest version of ROS 2 (Humble) was instrumented using LTTng tracepoints. The instrumentation was created to collect basic information about objects (i.e., publishers and subscriptions) and events (i.e., publications, timer and subscription callbacks). This was a first step, knowing that this might need to be improved or augmented when working on the actual analysis. Then, tools were created to integrate tracing into the ROS 2 tooling ecosystem, namely the command-line tools and launch system introduced in Section 2.4.1. To control LTTng tracing within the ROS 2 ecosystem, the LTTng bindings were used. Chapter 4 presents a framework with low-overhead instrumentation and tools for tracing ROS 2.

3.1.2 Trace Data Analysis

Using the Babeltrace Python bindings, a first trace processing prototype was created as a proof-of-concept for the initial instrumentation and corresponding execution models. Then, similar analyses were implemented using Trace Compass. Finally, the instrumentation and Trace Compass analyses were extended to be able to compute the message flow. Chapter 5 presents a method for extracting and displaying the path of a message across distributed ROS 2 systems.

3.2 Document Structure

This thesis adopts the recommended outline for a thesis by articles, where published or submitted articles are directly included as chapters. The structure is as follows:

- Chapter 1 introduces the research context and presents the research objectives.
- Chapter 2 summarizes relevant work in the literature.
- Chapter 3 explains the research approach that was used for this thesis.
- Chapter 4 presents a framework for tracing ROS 2, with instrumentation and tracing tools.
- Chapter 5 presents a method for visualizing the path of a message across distributed ROS 2 systems.
- Chapter 6 contains a general discussion following the work presented in Chapters 4 and 5.
- Chapter 7 concludes with a summary of the work presented in this thesis, its limitations, and possible future work.

CHAPTER 4 ARTICLE 1: ROS2_TRACING: MULTIPURPOSE LOW-OVERHEAD FRAMEWORK FOR REAL-TIME TRACING OF ROS 2

Preface

Full Citation: Christophe Bédard[1], Ingo Lütkebohle[2], and Michel Dagenais[1], “ros2_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2,” *IEEE Robotics and Automation Letters*, [accepted] 2022.

Abstract - Testing and debugging have become major obstacles for robot software development, because of high system complexity and dynamic environments. Standard, middleware-based data recording does not provide sufficient information on internal computation and performance bottlenecks. Other existing methods also target very specific problems and thus cannot be used for multipurpose analysis. Moreover, they are not suitable for real-time applications. In this paper, we present `ros2_tracing`, a collection of flexible tracing tools and multipurpose instrumentation for ROS 2. It allows collecting runtime execution information on real-time distributed systems, using the low-overhead LTTng tracer. Tools also integrate tracing into the invaluable ROS 2 orchestration system and other usability tools. A message latency experiment shows that the end-to-end message latency overhead, when enabling all ROS 2 instrumentation, is on average 0.0033 ms, which we believe is suitable for production real-time systems. ROS 2 execution information obtained using `ros2_tracing` can be combined with trace data from the operating system, enabling a wider range of precise analyses, that help understand an application execution, to find the cause of performance bottlenecks and other issues. The source code is available at: https://gitlab.com/ros-tracing/ros2_tracing.

Keywords - Software tools for robot programming, distributed robot systems, Robot Operating System (ROS), performance analysis, tracing.

Parts of this work were supported through funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 780785. Further, financial support of Ericsson, NSERC, and Prompt for C. Bédard and M. Dagenais is gratefully acknowledged.

[1]C. Bédard and M. Dagenais are with the Department of Computer Engineering and Software Engineering, Polytechnique Montréal, Montreal, Quebec H3T 1J4, Canada, `{christophe.bedard,michel.dagenais}@polymtl.ca`

[2]I. Lütkebohle is with Corporate Research at Robert Bosch GmbH, 71272 Renningen, Germany, `ingo.luetkebohle@de.bosch.com`

4.1 Introduction

As modern robots have become more versatile, e.g., in tackling unstructured environments or collaborative work, their software has become correspondingly more complex. Distributed, asynchronous compute graphs based on frameworks like ROS [82, 85] are now the dominant approach for integrated systems, even for space exploration [103]. Correspondingly, testing and debugging is now typically conducted by collecting data from a running system for later analysis, through tools like rosbag or textual logging [139, 140].

However, there are well-known drawbacks: rosbag and similar middleware-based tools can only record data that is available as messages. Aside from the effort involved, there is also a significant resource cost in both CPU and memory usage. It is also well known that perturbing a system through extensive monitoring is to be avoided [4]. Therefore, messages simply cannot practically deliver the stacktrace-level of detail and the detailed execution context information that we have come to expect from classical debuggers. Logging also cannot fill this gap, because of its unstructured output and lack of support for binary data. As a result, the debugging experience in robotics is greatly impoverished.

In contrast, tracing has been developed to provide structured, flexible, on-demand data capture across multiple applications and the kernel, to enable detailed analysis when needed. Conceptually, it can be considered an evolution of logging with support for binary data and well-defined data structures. Common frameworks provide support for easy and low-overhead capture of contextual data, such as process information or accurate, in-process timestamps, as well as aggregation of data across hosts and tooling for analysis [13, 28, 50, 58].

However, two challenges need to be solved to truly improve the testing and debugging situation. First, tracing has so far been a tool for performance experts, used in very specific analysis use-cases – such as scheduling optimization [128] or message latency analysis [148] – that were difficult to extend. To improve the debugging experience in general, a more versatile approach is required. Second, we need to ensure that the performance requirements of robotics are met. Standard ROS 2 targets *soft* real-time systems, i.e., systems where reaction time should have an upper bound, and should be achieved almost always, even though exceeding the bound is not catastrophic. To maintain these characteristics, a tracing integration must have comparatively low overhead with very few outliers.

Contributions. In this paper, we present `ros2_tracing`, a framework for tracing ROS 2 [85] with a collection of multipurpose low-overhead instrumentation and flexible tracing tools. This new tool enables a wider range of precise analyses that help understand an application execution. The source code is available at: https://gitlab.com/ros-tracing/ros2_tracing.

`ros2_tracing` brings the following contributions:

- It offers extensible tracing instrumentation capable of providing execution information on multiple facets of ROS 2.
- With a strategic two-phase instrumentation design and using a low-overhead tracer, it has a lower runtime overhead than current solutions, making it suitable for the real-time applications targeted by ROS 2.
- It enables more precise analyses using combined ROS 2 userspace and kernel space data as a whole.

Furthermore, notable `ros2_tracing` features include a close integration into the expansive suite of ROS 2 orchestration & general usability tools, and an easily swappable tracer backend to support different operating systems or to switch to a tracer with other desired features.

This paper is structured as follows: We survey related work in Section 4.2 and summarize relevant background information in Section 4.3. We then present our solution in Section 4.4 and discuss its analysis potential in Section 4.5. Thereafter, Section 4.6 presents an evaluation of the runtime overhead of our solution. Future work is outlined in Section 4.7. Finally, we conclude in Section 4.8.

4.2 Related Work

Tracing is an established approach for performance analysis, popular for operating system-level performance analysis and for distributed systems. Its popularity is both due to the low overhead when not in use, which is often zero, and due to the extensive tool support. An excellent overview, covering both cloud and operating system use-cases, is [4, 149]. However, while powerful, these tools arguably operate at an abstraction level that is too low to be practical for the average roboticist.

In the context of robotics and ROS 1, the earliest reported use of tracing was motivated by non-deterministic behavior of obstacle avoidance in a mobile robot, by one of the present authors [144]. Using a model of the ROS 1 navigation stack, and tracing based on LTTng [28], a lack of synchronization between the sensory data processing and motion control pathways could be identified. This is a primary example of how non-deterministic effects can be hard to diagnose otherwise, since the magnitude of the effect was dependent on how the OS scheduled the threads involved, which also varied over the run-time of the system. There were some attempts at generalizing this kind of tracing tooling in [145], and deriving a message flow analysis [146]. However, they were discontinued due to the emergence of ROS 2 and its potential for real-time applications [86, 94].

Previous work has identified various open problems in ROS 2. Kronauer et al. [119] investigated the end-to-end latency of communications and found that its overhead is up to 50% compared to directly using DDS, the underlying middleware. Similarly, Jiang et al. [118] found that the message conversion cost is highly dependent on the complexity of the message structure. Casini et al. [126] proposed a scheduling model that aims to bound the end-to-end latency of processing chains. Furthermore, several previous contributions propose tools that use tracing to measure and/or improve message transmission latency, due to its importance for realizing low-latency distributed systems. This includes the RAPLET tool by Nishimura et al. [148] for ROS 1, ROS-FM by Rivera et al. [147], and ROS-Llama by Blass et al. [128], both targeting ROS 2. The last two are monitoring tools, meaning that they use instrumentation to extract execution information, process it, and provide the results to users or act on them during runtime. However, none of these tools support any use-cases beyond latency, and they all show significant overheads (cf. Table 4.1), which is due for some of them to the use of logs or custom unoptimized tracers to extract execution information. An interesting custom proposal to detect latency deadline violations with a low overhead of only 86 µs per event, including online detection, is proposed by Peeck et al. [125], but again, without attempt at generality.

In conclusion, the present work is – to the best of our knowledge – the first and only tool that provides a generic, tracing-based approach for ROS 2 performance analysis.

4.3 Background

In this section, we summarize relevant background information needed to support subsequent sections. Note that we use “ROS 1” to refer to the first version of ROS and use “ROS” to refer to ROS 1 and ROS 2 in general, since many concepts apply to both.

4.3.1 ROS 2 Architecture

The ROS 2 architecture has multiple abstraction layers; from top to bottom, or user code to operating system: `rclcpp`/`rclpy`, `rcl`, and `rmw`. The user code is above and the middleware implementation is below. `rclcpp` and `rclpy` are the C++ and Python client libraries, respectively. They are supported by `rcl`, a common client library API written in C which provides basic functionality. The client libraries then implement the remaining features needed to provide the application-level API. This includes implementing executors, which are high-level schedulers used to manage the invocation of callbacks (e.g., timer, subscription, and service) using one or more threads. `rmw` is the middleware abstraction interface. Each middleware that is used with ROS 2 has an implementation of this interface. Multiple middleware implementations can be installed on a system, and the desired implementation can be selected at runtime through an environment variable, otherwise the default implementation is used. As of ROS 2 Galactic Geochelone, the default middleware is Eclipse Cyclone DDS [78].

4.3.2 ROS Nodes and Packages

ROS is based on the publish-subscribe paradigm and also supports the RPC pattern under the "service" name. ROS nodes may both publish typed messages on topics and subscribe to topics, and they can use and provide services. While the granularity and semantics of nodes in a system is a design choice, the resulting node and topics structure is analogous to a computation graph.

There are also specialized nodes called "lifecycle nodes" [93]. They are stateful managed nodes based on a standard state machine. This makes their life cycle easier to control, which can be beneficial for safety-critical applications [94]. Node life cycles can also be split into initialization phases and runtime phases, with dynamic memory allocations and other non-deterministic actions being constrained to the initialization phases for real-time applications.

As for the code, in ROS, it is generally split into multiple packages, which directly and indirectly depend on other packages. Each package has a specific purpose and may provide multiple libraries and executables, with each executable containing any number of nodes. The ROS ecosystem is federated: packages are spread across multiple code repositories, on various hosts, and are authored and maintained by different people. The ROS 2 source code itself is made up of multiple packages that approximately match its architecture.

4.3.3 Usability and Orchestration Tools

Much like ROS 1, ROS 2 has many tools for introspection, orchestration, and general usability. There are various `ros2` commands, including `ros2 run` to run an executable and `ros2 topic` to manually publish messages and introspect published messages. Packages can also provide extensions that add other custom commands. The `ros2 launch` command is the main entry point for the ROS 2 orchestration system and allows launching multiple nodes at once. This is configured through Python, XML, or YAML launch files which describe the system to be launched using nodes from any package or even other launch files. Since ROS systems can be quite complex and contain multiple nodes, an orchestration system is indispensable. Launch files can also be used to orchestrate test systems and verify certain behaviors or results.

4.3.4 Generalizability

Fig. 4.1 shows a summary of the ROS 2 architecture and the main tooling interaction. The architecture and launch system can be generalized down to an orchestration tool managing an application layer on top of a middleware. Therefore, the tool presented in this paper could be applied to other similar robotic systems.

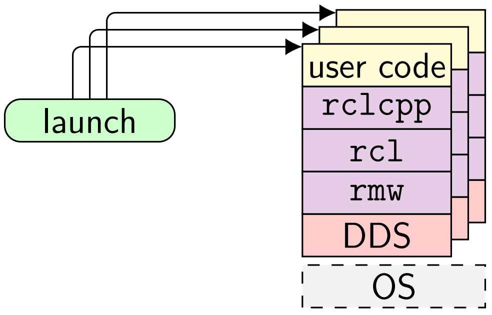


Figure 4.1 Overall ROS 2 architecture and tooling interaction.

4.4 `ros2_tracing`

In this section, we present the design and content of `ros2_tracing`. It contains multiple ROS packages to support three different but complementary functionalities: instrumentation, usability tools, and test utilities. Table 4.1 shows a comparison between our proposed method and the existing methods mentioned in Section 4.2.

4.4.1 Instrumentation

As shown in Fig. 4.2, core ROS 2 packages are instrumented with function calls to the `tracetools` package. This package provides tracepoints for all of ROS 2 and is the one that triggers them. Tracepoints are statements in the source code which allow capturing execution information using a hook mechanism [13]. They usually act as instrumentation points and could be directly added to the instrumented code. This creates an indirection, which we introduced for two complementary reasons. First, it allows for abstracting away the tracer backend and allows to easily switch the tracer. Indeed, by replacing the compiled `tracetools`

Table 4.1 Summary of Existing Monitoring and Instrumentation Methods and Comparison with Proposed Method

Method	**Type**[a]	**ROS 1 / 2**	**Instrumentation** messages	callbacks	services	executor	lifecycle	**Extensible**	**Launch tools**	**Overhead**[b]	**Source avail.**
ROS-FM [147]	M	1 / 2	✓	×	✓	- / ×	- / ×	×	×	15-515% C	×
tracetools [145,146]	I	1	✓	✓	×	-	-	×	×	?	✓
RAPLET [148]	I	1	✓	✓	×	-	-	×	×	2-20% L	✓
ROS-Llama [128]	M	2	✓	✓	×	✓	×	×	×	30-40% C	×
`ros2_tracing`	I	2	✓	✓	✓	✓	✓	✓	✓	1-15% L	✓

[a] M and I for monitoring and instrumentation-only types, respectively.
[b] C and L for CPU and latency overhead, respectively.

library, the tracer backend can be replaced without affecting the instrumented packages (i.e., the core ROS 2 code). This could be done to support tracing on a different platform or to use a tracer that has other desired functionalities. Second, this keeps instrumented packages free of boilerplate code (e.g., tracepoints definitions and other required preprocessor macros). However, the main advantage of this design choice also has a slight downside, since adding new tracepoints requires modifying both the instrumented package and the `tracetools` package.

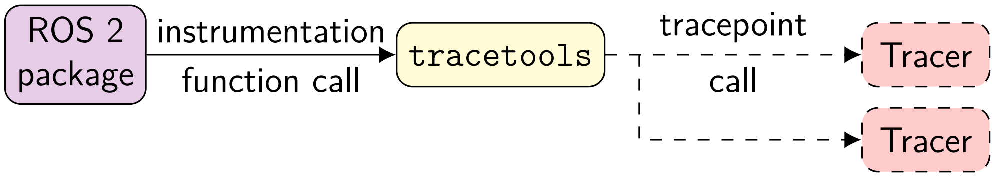


Figure 4.2 Instrumentation and tracepoint calls.

As shown in Table 4.1, in addition to being extensible, our method provides instrumentation for multiple aspects of ROS 2, including messages, callbacks, services, executor states, and lifecycle states. Table 4.2 presents a full list of the instrumentation points provided by `tracetools`. We split the instrumentation points into two types: initialization events and runtime events. The former collect one-time information about the state of objects, e.g., creation of publishers, subscriptions, and services. The latter collect information about events throughout the runtime, e.g., message publication and callback execution. The former are therefore predominantly triggered during the system initialization phase and are used to minimize the payload size of the latter. This strategic instrumentation design is key to minimizing the overhead in the runtime phase. For example, all publisher-related tracepoints in the runtime phase include a unique identifier for the publisher, which is then matched with the data collected by the publisher-related tracepoints in the initialization phase (e.g., topic name, corresponding node name, etc.), thus minimizing the payload size of runtime tracepoints. The information that is collected to form the trace data can then be used to build a model of the execution. Due to the very abstractional nature of the ROS 2 architecture, multiple instrumentation points are sometimes needed to gather the necessary information. For example, to build a model of a subscription, we collect information about its callback function from `rclcpp` and its topic name from `rcl`. This is because callbacks are managed by the client library. Furthermore, both the instrumentation point name and the payload are meaningful: some instrumentation points only differ by their names and are used to indicate the originating layer. Since the instrumentation points cover multiple analysis use-cases, if a portion of the information is not needed for a given analysis, some of the instrumentation points can be disabled and therefore have virtually no impact on execution.

We did not instrument the Python client library, since it is not used for the kind of real-time applications that we are considering with `ros2_tracing`. However, we instrumented `rcl` directly whenever possible. By putting the instrumentation as low as possible in the ROS 2 architecture, it can be leveraged to more easily support other client libraries in the future.

The *Linux Trace Toolkit: next generation* (LTTng) tracer was chosen as the default tracing backend, for its low overhead and real-time compatibility as well as its ability to trace both the kernel and userspace [28, 38]. The runtime cost per LTTng userspace tracepoint on a vanilla Linux kernel using an Intel i7-3770 CPU (3.40 GHz) with 16 GB of RAM is approximately 158 ns [13]. Since it is a Linux-only tracer, all instrumentation calls to the `tracetools` package are preprocessed out on other platforms. This can also be achieved on Linux through a build option.

4.4.2 Usability Tools

In line with the ROS 2 usability and orchestration tools, our proposed solution includes two different interfaces to control tracing: a `ros2 trace` command and a `Trace` action for launch files. The `ros2 trace` command is a simple command that allows configuring the tracer to start tracing. The system or executable to be traced must then be run or launched in a separate terminal. When the application is done running, tracing must be stopped in the original `ros2 trace` terminal. On the other hand, the `Trace` action can be used in XML, YAML, and Python launch files. It then allows configuring the tracer and launching the system at the same time. Tracing is stopped automatically after the launched system has shut down, either on its own or after being manually terminated. Listing 4.1 shows an example with an XML file that launches two nodes. While ROS 2 does not currently natively

Table 4.2 Instrumentation Points List with Types

ROS 2 Layer	**Instrumentation Point Name**	**Type**[a]	**Note**[b]
`rclcpp`	rclcpp_subscription_init	I	
	rclcpp_subscription_callback_added	I	
	rclcpp_publish	R	P
	rclcpp_take	R	S
	rclcpp_service_callback_added	I	
	rclcpp_timer_callback_added	I	
	rclcpp_timer_link_node	I	
	rclcpp_callback_register	I	
	callback_start	R	S
	callback_end	R	
	rclcpp_executor_get_next_ready	R	S
	rclcpp_executor_wait_for_work	R	S
	rclcpp_executor_execute	R	S
`rcl`	rcl_init	I	
	rcl_node_init	I	
	rcl_publisher_init	I	
	rcl_subscription_init	I	
	rcl_publish	R	P
	rcl_take	R	S
	rcl_client_init	I	
	rcl_service_init	I	
	rcl_timer_init	I	
	rcl_lifecycle_state_machine_init	I	
	rcl_lifecycle_transition	R	
`rmw`	rmw_publisher_init	I	
	rmw_subscription_init	I	
	rmw_publish	R	P
	rmw_take	R	S

[a] I and R for initialization and runtime types, respectively.
[b] P and S for publishing and message reception hot paths, respectively.

support it, this would be useful for remote or multi-host orchestration to trace all hosts at once and aggregate the resulting traces.

Listing 4.1 `Trace` action in XML launch file

```
<launch>
  <trace
    session-name="ros2" events-ust="ros2:*"/>
  <node pkg="package_a" exec="executable_x"/>
  <node pkg="package_b" exec="executable_y"/>
</launch>
```

Furthermore, these tools can be used to leverage existing LTTng instrumentation (e.g., kernel and other userspace instrumentation) and to enable any custom application-level tracepoints to record other relevant data. For example, LTTng provides shared libraries that can be preloaded with `LD_PRELOAD` to intercept calls to `libc`, `pthread`, the dynamic linker, and function entry & exit instrumentation (added with `-finstrument-functions`) and trigger tracepoints before calling the real functions. If those tracepoints are enabled through launch files, the corresponding shared libraries will be located and preloaded automatically for all executables, which greatly simplifies the launch configuration. The tools also do not prevent users from directly configuring the tracer for advanced options. They are only a thin flexible compatibility and usability layer for ROS 2 and use the LTTng Python bindings for tracer control.

4.4.3 Test Utilities

The `tracetools_test` package provides a test utility that allows running nodes and tracing them. The resulting trace can then be read in the test using the `tracetools_read` package to assert results or behaviors in a lower-level, less invasive way.

4.5 Analysis

The instrumentation and tracing tools provided by `ros2_tracing` allow collecting execution information at the ROS 2 level. This information can then be processed using existing tools to compute metrics or to provide models of the execution. For example, we traced a ROS 2 system that simulates a real-world autonomous driving scenario [100]. In this example, a node receives data from 6 different topics using subscriptions. When the subscriptions receive a new message, they cache it so that the node only keeps the latest message for each topic. The periodically-triggered callback uses the latest message from each topic to compute a result and publish it. To analyze the trace data, we wrote a simple script using `tracetools_analysis` [150], a Python library to read and analyze trace data. As shown in Fig. 4.3, we can display message reception and publication instance timestamps as well as timer callback execution intervals over time. There is a visible gradual time drift between the input and output messages, which could impact the validity of the published result, similar to the issue described by [144]. This could warrant further analysis and tuning, depending on the system requirements. We can also compute and display the timer callback execution interval and duration, as shown in Fig. 4.4. The timer callback period is set to 100 ms and the duration is approximately 10 ms, but there are outliers. This jitter could negatively affect the system; these anomalies could warrant further analysis as well.

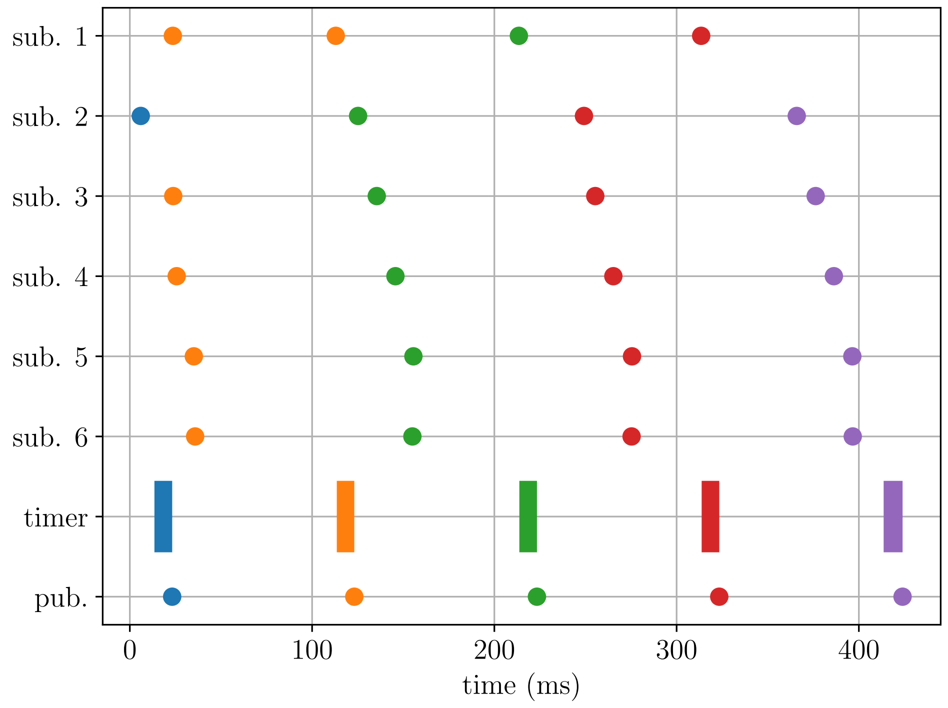


Figure 4.3 Example time chart of subscription message reception (sub.), timer callback execution (timer), and message publication (pub.). Message reception and publication instances are displayed as single timestamps, while timer callback executions are displayed as ranges, with a start and an end. The periodic timer callback uses the last received message from each subscription to compute a result and publish it; this inputs-outputs link is illustrated using colors, highlighting an inadequate synchronization.

To dig deeper, this information can be paired with data from the operating system: OS trace

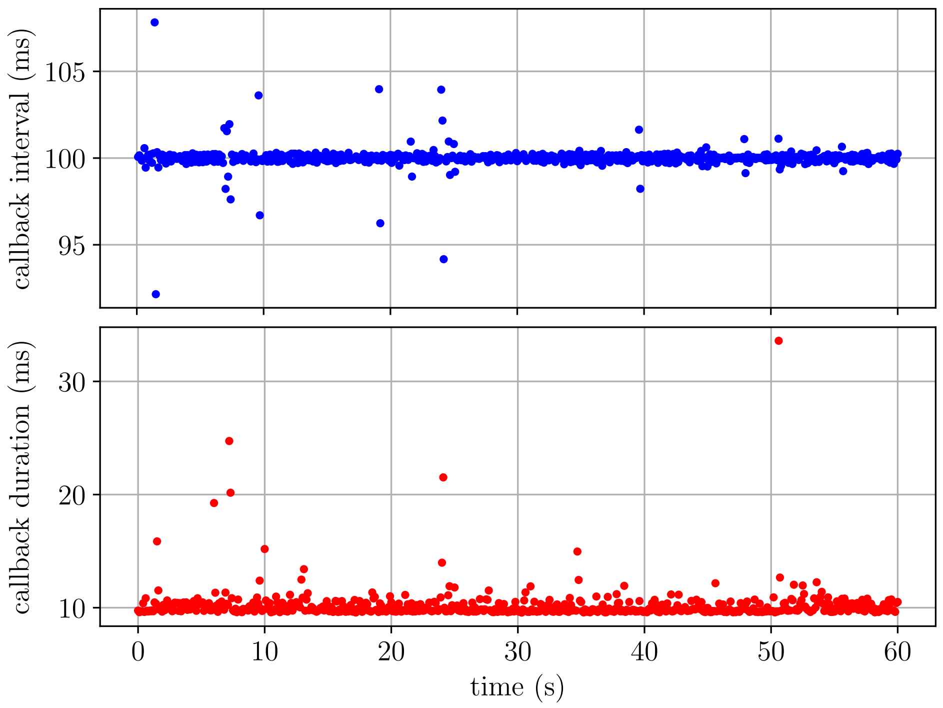
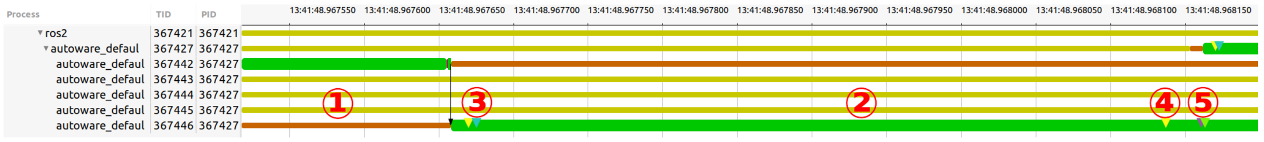


Figure 4.4 Example timer callback execution interval (top) and duration (bottom) over time. The callback period is set to 100 ms, while the callback duration depends on the work done. Both contain outliers.



Process	TID	PID
▾ ros2	367421	367421
▾ autoware_defaul	367427	367427
autoware_defaul	367442	367427
autoware_defaul	367443	367427
autoware_defaul	367444	367427
autoware_defaul	367445	367427
autoware_defaul	367446	367427

Figure 4.5 State of ROS 2 application threads over time with timestamps of ROS 2 events from the `ros2_tracing` instrumentation displayed as small triangles on top of the thread state rectangle: ① thread waiting for CPU for 9.9 ms (orange), ② thread running (green), ③ event marking start of middleware query & wait for new messages, ④ event marking end of middleware query, and ⑤ `rclcpp_executor_execute` event followed shortly after by `callback_start` event for timer callback. This result was obtained by importing trace data collected from the Linux kernel and from the ROS 2 application using LTTng into Trace Compass [1]. The black arrow to the left of ③ represents the scheduling switch from one thread to another for a given CPU. Some less relevant threads were hidden.

data can help find the cause of performance bottlenecks and other issues [55]. Since ROS 2 does higher-level scheduling, this is critical for understanding the actual execution at the OS level. Using LTTng, the Linux kernel can be traced alongside the application. The ROS 2 trace data that was obtained using the `ros2_tracing` instrumentation can be analyzed together with the OS trace data using Eclipse Trace Compass [1], which is an open-source trace viewer and framework aimed towards performance and reliability analysis. Trace Compass can analyze Linux kernel data to show the frequency and state of CPUs over time, including interrupts, system calls, or userspace processes executing on each CPU. It can also display the state of each thread over time, as shown in Fig. 4.5 for the application threads. Building Trace Compass analyses specific to ROS 2 is left as future work; however, we can visualize timestamps of ROS 2 trace events on top of the existing analyses. From the ROS 2 trace data shown in Fig. 4.4, we know that the timer callback instance following the longest interval is at the 1.4 s mark with 107.8 ms. Finding the timestamps of the corresponding ROS 2 events in Trace Compass, we see that the thread was blocked waiting for CPU for 9.9 ms before the aforementioned callback instance. The thread then had to query the middleware for new messages (even if timers are strictly handled at the ROS 2 level) and finally call the overdue timer callback. In this example, a multi-threaded executor was used, with the number of threads being equal to the number of logical CPU cores by default. Since this was not the only application running on the system at that time, multiple processes and threads were competing for CPU, as can be observed using Trace Compass. Therefore, the executor settings could be tuned, or the executor could be replaced by another type of executor with features that better meet the requirements for this system, which could entail creating a new executor: this is an open problem in ROS 2. A multi-level analysis such as this one would not have been possible without collecting both userspace & kernel execution information, and analyzing the combined data, which current tools do not offer. The scripts and full instructions to replicate this example are available at: github.com/christophebedard/ros2_tracing-analysis-example.

4.6 Evaluation

To validate that our proposed solution is compatible latency-wise with real-time systems, we evaluate the overhead of `ros2_tracing`, or specifically its instrumentation overhead. This is the time consumed by the instrumentation within the monitored process. When enabled, it directly affects these processes by adding latency.

Since the `ros2_tracing` tracepoints are placed along the message publication and reception pipeline, an easy way to capture the maximum overhead is to measure the time between publishing a message and when it is handled by the subscription callback. This is what we will do in the following.

4.6.1 Experiment Setup

We use the standard message-passing latency benchmark for ROS 2, performance_test [107], with a minimal configuration: one publisher node and one subscription node. We vary message size and publishing rate, since it is known that they affect middleware performance [119]. The parameter space is shown in Table 4.3 and is based on typical use-cases [101].

Table 4.3 Experiment Parameters and Values

Publishing rate (Hz)	100, 500, 1000, 2000
Message size (KiB)	1, 32, 64, 256
Quality of service	reliable only
DDS implementation	eProsima Fast DDS

To reduce measurement variability, we follow common practice by using a kernel patched with the PREEMPT_RT patch (5.4.3-rt1), disabling simultaneous multithreading, and disabling power-saving features in the BIOS (dynamic frequency scaling, C-states, Turbo Boost, etc.). Further, we increase UDP buffers to 64 MB to ensure sufficient networking performance for larger messages. The experiment was run on an Intel i7-3770 (3.40 GHz) 4-core CPU, 8 GB RAM system with Ubuntu 20.04.2. All measurements are based on the ROS 2 Rolling distribution, in between the Galactic and Humble releases, which is the most recent version at this time. While Eclipse Cyclone DDS [78] is the default DDS implementation for the current ROS 2 release, Galactic, we have found eProsima Fast DDS [79] to be more stable for larger messages, and it is also the default for the upcoming release, Humble. We have therefore used Fast DDS.

For each combination in the parameter space, performance_test is run for 60 minutes and scheduled with the SCHED_FIDO real-time policy with the highest priority (99). To reduce outliers due to system initialization, the first 10 seconds of each recording are discarded. To determine the tracing overhead, we run the experiment once without any tracing enabled, and once with all tracepoints enabled. In practice, since not all tracepoints might be needed depending on the intended analysis, this represents the worst-case scenario.

The code and instructions to replicate this experiment are available at: github.com/christophebedard/ros2_
overhead-evaluation.

4.6.2 Results and Discussion

Fig. 4.6 shows the individual average latencies without and with tracing, while Fig. 4.7 shows the absolute and relative latency overhead.

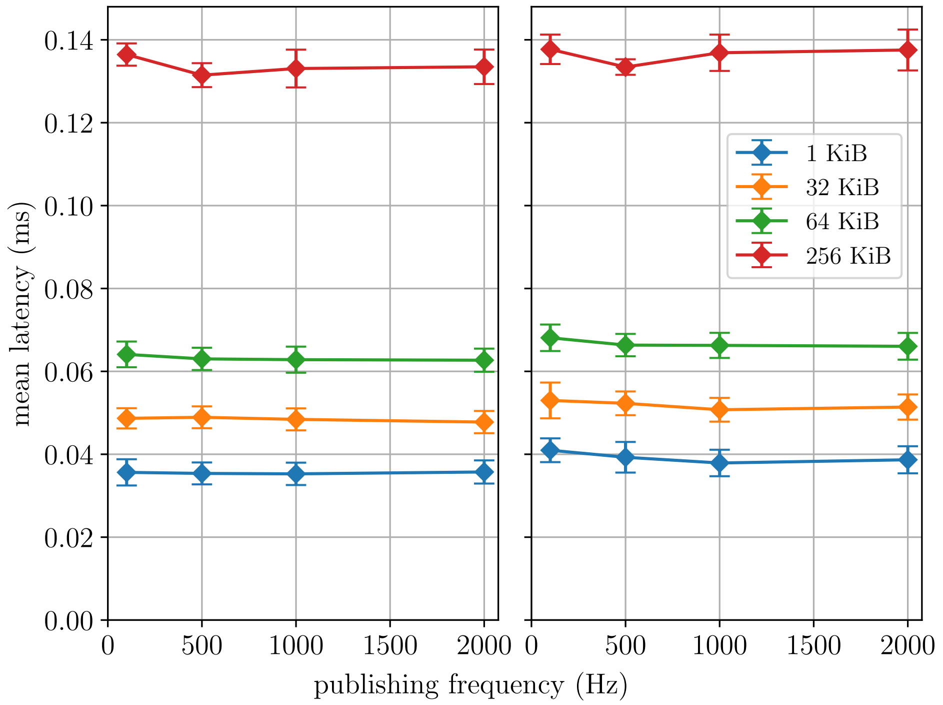


Figure 4.6 Message latencies (avg. ± std.) without tracing (left) and with tracing (right).

First, as expected, the mean latency values increase with the message size, and the relative latency overhead values decrease with the message size. There is no significant decrease in latency as the publishing frequency increases; this behavior was however much more notice-

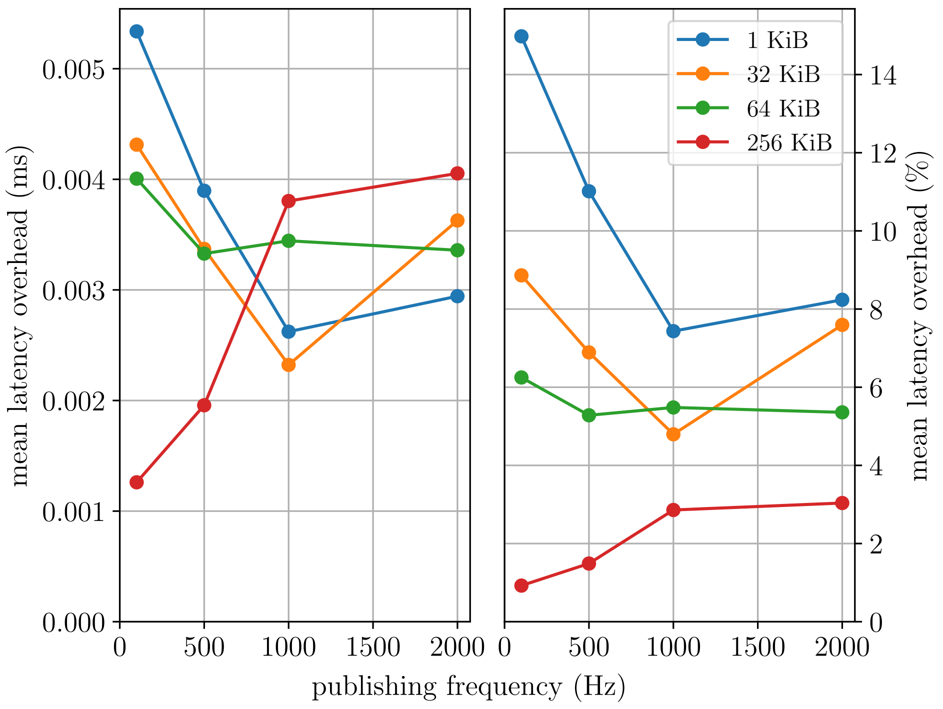


Figure 4.7 Absolute (left) and relative (right) latency overhead results. The standard deviation of the difference between the two means is insignificant here.

able before disabling power-saving features through the BIOS. We would also expect the CPU overhead to be the same for all message sizes and publishing frequencies, since it is, in theory, a constant overhead for message publication and reception. However, it can be seen that this is not the case, and instead, the absolute overhead is larger at small frequencies. This is somewhat puzzling, and it would certainly merit further experiments. However, due to the overall small effect, we are approaching a range where cache effects in the CPU or other non-deterministic factors come into play. Since the overhead values are overall fairly close, the overhead does not seem to be related to any of the experiment parameters, and the absolute values are well within acceptable ranges, we consider the requirements set out for `ros2_tracing` to be fulfilled. Additionally, these absolute latency overhead results are within one order of magnitude of the results that [13] presented: since there are 10 tracepoints in the publish-subscribe hot path (see Table 4.2), the overhead should therefore be $10 \cdot 158$ ns $= 0.00158$ ms, which is indeed comparable.

Since most practical systems use a mixture of message sizes and frequencies, we also analyze the overhead by aggregating it over all experiment runs. For each combination of message size and publishing frequency, we use the two sets of latencies (i.e., without tracing and with tracing, represented in Fig. 4.6), and subtract the mean of the no-tracing set from all latencies. By aggregating the latency differences for all combinations, we obtain two sets of latency overheads, which are represented in Fig. 4.8 (without tracing and with tracing, respectively). Note that the aggregate overhead is more strongly influenced by the higher publication frequencies, since more messages are sent in the same time frame. The mean overhead is thus 0.0033 ms, with 50% of the data between 0.0010 ms and 0.0056 ms.

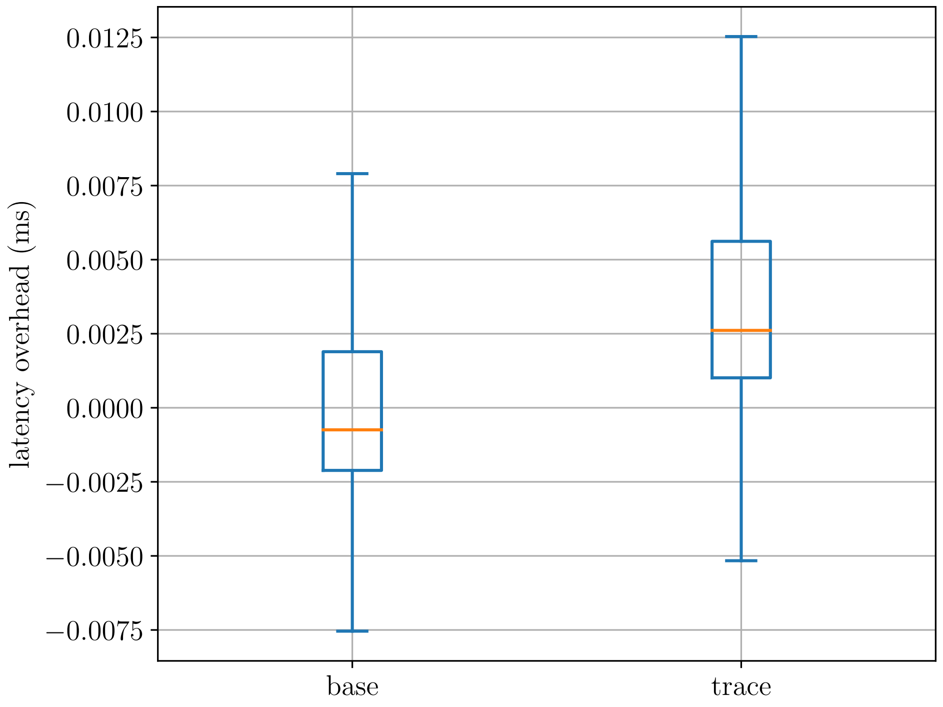


Figure 4.8 Aggregated latency overhead and variation without tracing (left) and with tracing (right). Latency values have been individually normalized to zero mean based on the latencies without tracing, showing overhead and variation. Note that the left mean is very slightly below zero due to the additional imbalance caused by the variation in publishing frequency.

No measurement system can be entirely without overhead – however, we think that these results show that `ros2_tracing` has very low overhead, which is acceptable for most target situations for ROS 2. It is certainly lower than known alternatives as presented in Table 4.1.

Nonetheless, for very busy systems or very particularly CPU-constrained platforms, overheads might add up enough to impact the messaging performance. For such situations, there are two potential optimization options: First, about half of the tracepoints would be sufficient for basic information, and, second, tracing can be selectively enabled on only some of the processes, instead of all ROS 2 nodes.

4.7 Future Work

Many improvements and additions could be made to `ros2_tracing`. While the RPC pattern is not used as much in real-time applications, instrumentation could be added to support services and actions, with the latter being services with optional progress feedback. Furthermore, `ros2_tracing` does not gather information about object destruction, e.g., if a publisher or a subscription is destroyed during runtime. This is because it does not fit with design guidelines of real-time safety-critical systems, where the system is usually static once it enters its runtime phase. Nonetheless, complete object lifetime information could be gathered. Middleware implementations could also be instrumented to provide lower-level information on the handling of messages. Instrumenting `rclc` [134, 135] would also be interesting. It is a client library written in C with a deterministic executor aimed at ROS 2 applications on memory-limited real-time platforms such as microcontrollers.

As for the usability tools, as mentioned previously, the orchestration tools could be improved, when native support for remote or coordinated multi-host orchestration gets added to the ROS 2 launch system.

While the overall approach taken for `ros2_tracing` is primarily aimed at offline monitoring and analysis, the instrumentation itself could be leveraged for online monitoring. For example, the LTTng live mode could be used to do online processing of the trace data. Another backend could also be added for a tracer that better supports online monitoring. There could also be other default backends for other operating systems, like QNX, which is often used for real-time as well.

In future work, we plan on building on the `ros2_tracing` instrumentation and tools to analyze the internal workings of ROS 2. For example, as mentioned in Sections 4.2 and 4.5, the determinism and efficiency of the ROS 2 executors could be analyzed and compared to proposed alternatives. The ROS 2 instrumentation could of course also be used in conjunction with the LTTng built-in userspace and kernel instrumentation, as demonstrated in Section 4.5. For example, to verify real-time systems, unwanted runtime phase dynamic memory allocations could be detected by combining lifecycle node state information and the LTTng `libc` memory allocation tracepoints. Trace Compass could also be used to provide analyses and views specific to ROS 2.

4.8 Conclusion

Testing and debugging robotic systems, based on recordings of high-level messages, does not provide sufficient information on the computation performed, to identify causes of performance bottlenecks or other issues. Existing methods target very specific problems and thus cannot be used for multipurpose analysis. They are also not suitable for real-world real-time applications, because of their high overhead or poor usability.

We presented `ros2_tracing`, a framework with instrumentation and flexible tools to trace ROS 2. The extensible multipurpose low-overhead instrumentation for the ROS 2 core allows collecting execution information to analyze any ROS 2 system. The tools promote usability through their integration with the ROS 2 orchestration system and other usability tools. Our experiments showed that the message latency overhead it introduces is in an acceptable range for real-time systems built on ROS 2. These tools enable testing and debugging ROS 2 applications based on internal execution information, in a manner that is compatible with real-time applications and real-world development processes. Analyzing the combined trace data, from a ROS 2 application and the operating system, can help find the cause of performance bottlenecks and other issues. We plan on leveraging `ros2_tracing` in future work to analyze the internal handling of ROS 2 messages.

CHAPTER 5 ARTICLE 2: MESSAGE FLOW ANALYSIS WITH COMPLEX CAUSAL LINKS FOR DISTRIBUTED ROS 2 SYSTEMS

Preface
Full Citation: Christophe Bédard[1], Pierre-Yves Lajoie[1], Giovanni Beltrame[1], and Michel Dagenais[1], "Message Flow Analysis with Complex Causal Links for Distributed ROS 2 Systems," *Robotics and Autonomous Systems*, [in review] 2022.

Abstract - Distributed robotic systems rely heavily on publish-subscribe frameworks such as the Robot Operating System (ROS) to efficiently implement modular computation graphs. The ROS 2 executor, a high-level task scheduler which handles ROS 2 messages, is a performance bottleneck. We extend `ros2_tracing`, a framework with instrumentation and tools for real-time tracing of ROS 2, with the analysis and visualization of the flow of messages across distributed ROS 2 systems. Our method detects one-to-many and many-to-many causal links between input and output messages, including indirect causal links through simple user-level annotations. We validate our method on both synthetic and real robotic systems, and demonstrate its low runtime overhead. Moreover, the underlying intermediate execution representation database can be further leveraged to extract additional metrics and high-level results. This can provide valuable timing and scheduling information to further study and improve the ROS 2 executor as well as optimize any ROS 2 system. The source code is available at: github.com/christophebedard/ros2-message-flow-analysis.

Keywords - Software tools for robot programming, distributed robot systems, Robot Operating System (ROS), performance analysis, tracing.

5.1 Introduction

Modern robotic systems often leverage complex distributed processing: they use distributed perception [151], motion planning [152], and decision making [153]. They are built on software frameworks like ROS 2 [85], the successor to the Robot Operating System (ROS) [82]. Such publish-subscribe frameworks greatly simplify the development of modular computation graphs. However, high-level scheduling of tasks in ROS 2 (i.e., internal message handling,

The financial support of Ericsson, NSERC, Prompt, and Vanier Canada Graduate Scholarship is gratefully acknowledged.

[1]Department of Computer Engineering and Software Engineering, Polytechnique Montréal, Montreal, Quebec H3T 1J4, Canada, `{christophe.bedard, pierre-yves.lajoie, giovanni.beltrame, michel.dagenais}@polymtl.ca`

and subscription, service, or timer callback execution) brings a number of performance challenges. Several methods and tools have been proposed to study the default ROS 2 executor and compare its performance with other proposed executor designs [132–135]. However, such techniques are either not applicable to existing ROS 2 systems, since they require code modifications, or result in significant runtime overhead.

Low-overhead tracing has been used as a way to extract execution information for performance analysis purposes without impacting or perturbing the system. Furthermore, various techniques allow combining and correlating kernel and userspace events from multiple traces (i.e., from distributed systems). In previous work, we proposed `ros2_tracing` [154], a framework with low-overhead instrumentation and orchestration tools for tracing ROS 2. This tracing framework allows extracting ROS 2 execution information, and can be extended with additional instrumentation for more advanced use-cases or performance analysis goals.

Moreover, critical path analysis has been used to model the interactions inside parallel and distributed systems. For example, wait-related kernel events can be used to recursively compute wait dependencies across machines for requests spanning multiple hosts in a distributed system [66]. This technique helps to understand and explain the actual process execution, and to identify the prime target for performance optimization (i.e., the bottleneck). Likewise, in this paper, we present a message flow analysis method for ROS 2 distributed computation graphs. It can be useful for both end users of ROS 2, looking to analyze & optimize their system, and for developers, looking to do the same for the internals of ROS 2, although these two target audiences are not necessarily distinct.

Contributions. As shown in Fig. 5.1, our proposed technique can extract and visualize the paths of messages across distributed ROS 2 systems. Building a message flow graph in a low-overhead way, without modifying the applications themselves as the existing methods do, requires more complex execution data collection and analysis. With this novel approach, we bring the following contributions:

- An intermediate execution representation database of trace data obtained from a distributed system, providing information on ROS 2 objects (i.e., nodes, publishers, subscriptions, and timers) and events (i.e., message publication & reception instances, and subscription & timer callback instances). This database can be further leveraged to derive additional metrics and high-level results.

- Matching of published and received messages, compatible with distributed systems, without needing to modify the user code and without adding significant overhead.

- Inference of one-to-one and one-to-many causal links between received messages and

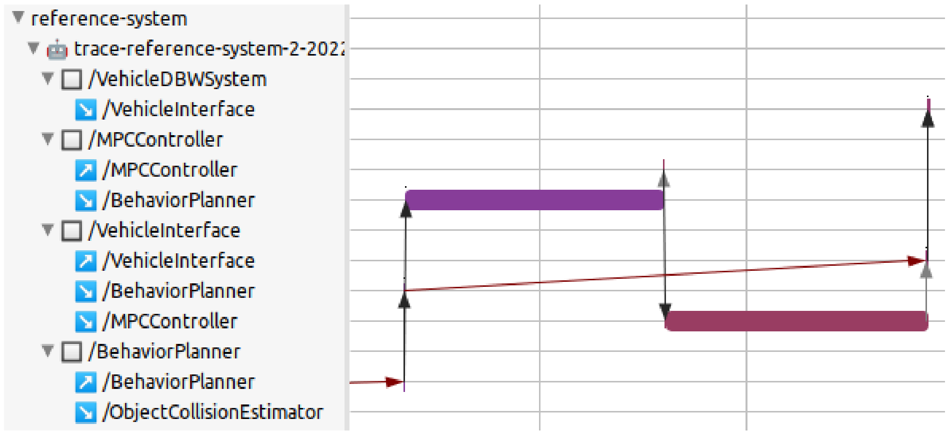


Figure 5.1 Message flow visualization using our method.

published messages, both automatically for direct links, and using simple user-level annotations for more complex indirect causal links.

- Extraction and visualization of the flow of messages across distributed ROS 2 systems, as well as the state of the executor over time for each process.
- Experiments and validation of our proposed method on both synthetic and real robotic systems, demonstrating that it can be used to study and optimize existing systems, and that it has a low runtime overhead.

This paper is structured as follows: We first survey related work in Section 5.2 and summarize relevant background information in Section 5.3. We then describe our intermediate execution representation in Section 5.4 and present our analysis method in Section 5.5. Thereafter, Section 5.6 presents experiments where we apply our method to two systems, and Section 5.7 provides an evaluation of the runtime overhead. Future work is outlined in Section 5.8. Finally, we conclude in Section 5.9.

5.2 Related Work

Previous work has identified open problems relating to the communication latency of ROS 2 [85] and its executor. Relevant methods were proposed to observe and study those problems.

5.2.1 Communications

First, the general performance of ROS 2 was evaluated by Maruyama *et al.* [109], Gutiérrez *et al.* [110], and Puck *et al.* [111]. Other work focuses on more specific elements of the performance of ROS 2, including its overhead with relation to the underlying middleware, DDS [75]. Kronauer *et al.* [119] evaluated the overhead of ROS 2 using profiling, and showed that it can lead to a 50% latency overhead. Some of this overhead can be attributed to the serialization and deserialization of complex message structures. Jiang *et al.* [118] proposed an adaptive serialization technique to improve communication performance by up to 93%. Wang *et al.* [117] proposed a single-host inter-process communications (IPC) layer for ROS 1 [82] and ROS 2 which reduces the overhead of IPC for large messages. Finally, Puck *et al.* [120] noted in another performance evaluation of ROS 2 that the use of dynamic memory allocations, when fetching new messages from the underlying middleware, accounts for a significant portion of the internal message processing time.

Various tools have been proposed to study message latency in ROS 2. The performance_test [107] benchmarking tool allows measuring the latency between publishers and subscriptions directly, while [108] allows defining a custom message graph topology. However, these benchmarking tools only evaluate the performance of a synthetic system or communication configuration. To measure the performance of real systems, observability tools are needed. Nishimura *et al.* [148] proposed RAPLET, which breaks down the latency between the publication of a message and the execution of the subscription callback function on the other end. It tracks messages using a sequence number in the message structure itself. Unfortunately, this sequence number field is not included in all messages. Similarly, Witte and Tichy [143] presented a tool to track messages in ROS 1 in order to interactively modify their content. These techniques cannot be applied to existing systems, since they require the addition of a custom message header, and their runtime overhead is significant, which can affect the validity of their results [4].

5.2.2 Executor

Moreover, other previous work has identified and studied open problems with the ROS executor, which is a high-level task scheduler [97]. It is responsible for fetching new messages from the underlying middleware and executing the corresponding subscription callbacks as well as timer callbacks, making the executor a clear performance bottleneck. Furthermore, scheduling tasks on top of the OS scheduler itself is challenging. Multiple methods model exchanges of ROS messages, from node to node, as event chains and pipelines in a directed acyclic graph (DAG). Peeck *et al.* [125] focused on online monitoring for reacting to latency violations in event chains. Casini *et al.* [126] proposed a formal scheduling model and a response-time analysis for ROS 2 to bound worst-case response times. Tang *et al.* [127] then proposed a more specific version that is, however, only valid for independent linear processing chains. Blass *et al.* [128] built on the work by Casini *et al.* [126] and proposed an online automatic latency manager for ROS 2. Their work also helped illustrate how the higher-level scheduling of tasks in ROS 2 does not interact well with classic OS-level scheduling techniques. Blass *et al.* [129] further extended this work, and stressed how the ROS 2 executor differs from normal schedulers in the literature, since it inherently prioritizes in order: timers, subscriptions, service servers, and service clients.

To help tackle some of these challenges, new executor designs have been proposed for ROS 2. The callback-group-level executor [132], now available as an alternative in ROS 2, allows having multiple distinct executor instances on multiple threads without interference. This enables scheduling of the OS threads themselves, using different priorities depending on system requirements, instead of bundling all ROS 2 elements together, as the default executor does. This results in lower latencies for higher-priority callback groups, as demonstrated by Yang and Azumi [131]. Similarly, Choi *et al.* [133] proposed a priority-driven chain-aware scheduler and showed that it helps lower end-to-end latencies as well. Staschulat *et al.* [134, 135] proposed a budget-based executor for real-time operating systems. To benchmark and compare executor designs, a reference system was proposed [100]. It is based on the computation graph of Autoware [98,99], an autonomous driving system completely based on ROS.

5.2.3 Tracing and Data Analysis

To investigate performance issues, low-overhead tracing has been widely used for collecting execution information in a minimally-invasive way. In particular, the LTTng tracer [28], which has a low runtime overhead [13], was used by Lütkebohle [144] to investigate determinism and message timing issues in ROS 1. They proposed [145] as a generic tracing tool for ROS 1. As a follow-up to this for ROS 2– and to improve on it – in previous work, we presented `ros2_tracing` [154], a framework with low-overhead instrumentation and tracing tools for ROS 2. The proposed instrumentation allows extracting simple metrics, such as publishing rate and subscription or timer callback duration. For more advanced use-cases, the instrumentation can easily be extended.

To extract useful information from trace data, advanced trace analysis methods build models from the trace data. One interesting technique is the critical path method, where the critical path is defined as the longest path in a DAG. The critical path is therefore the overall program or end-to-end latency bottleneck; shortening it effectively reduces the total execution time. Yang and Barton [62] applied the method to compute the critical path of the execution of parallel and distributed programs. Trace data from multiple hosts in a distributed system can be combined and synchronized for analysis as a whole [49–51]. Giraldeau and Dagenais [66] used wait-related trace events from the kernel (e.g., scheduling, network, or interrupts) to recursively compute wait dependencies across machines. While such wait-related operating system primitives are used in many applications, application-level information is required for more specialized analyses [58]. For example, ROS-level information could be used to apply the critical path method to the computation graph of a ROS system.

Previous work has partially tackled this critcial path analysis effort. Li *et al.* [155] used `ros2_tracing` [154] and `tracetools_analysis` [150], a simple trace data processing library, to provide an end-to-end latency breakdown. However, links between input subscriptions and output publishers need to be manually provided by the user; they are not automatically detected. [145] was used and extended by [146] to visualize the flow of messages. Unfortunately, [146] has many limitations and uses simplistic assumptions which do not always hold. For example, to track messages between nodes, it selects the first TCP packet that is queued after a message is sent by ROS 1. It then matches that network packet when it is received on the other end, and selects the next message reception event. This heuristic is simple and does not require adding additional fields to the messages themselves, but it is far from solid, since it could select packets from other applications. Furthermore, it only considers direct causal links inside subscription callbacks, i.e., where the message being processed by the callback instance is linked to the message that is published during that callback instance. However, many systems use custom message cache mechanisms that are independent from the ROS 2 API, which makes detecting and modeling those causal links far from trivial. Finally, it does not support one-to-many or many-to-many causal links, i.e., where one or more input messages are linked to more than one output message, and also does not work with more than one machine.

5.2.4 Summary

In summary, numerous latency- and executor-related open problems exist in ROS 2. Building a model and graph of the path of messages across a ROS 2 system would provide useful information to further study or work on resolving those open problems. Thus, we propose a low-overhead technique that can transparently & natively track messages while supporting complex application-dependent causal links between messages.

5.3 Background

In this section, we summarize relevant information required to support subsequent sections.

ROS 2 contains multiple abstraction layers. From top to bottom, i.e., from user-level to OS-level: `rclcpp` & `rclpy`, `rcl`, and `rmw`. The client libraries, `rclcpp` and `rclpy`, offer the actual user-facing ROS 2 C++ and Python APIs, respectively. They use a common underlying library, `rcl`; this architecture reduces duplicate code and thus makes adding new client libraries simpler. Then, `rcl` calls `rmw`, the middleware interface. This interface is implemented for each underlying middleware implementation, e.g., for each distinct DDS implementation. This allows ROS 2 to use any message-passing middleware, as long as it is done through this interface.

5.4 Intermediate Execution Representation

Before extracting the flow of ROS 2 messages across a distributed system, we first process the raw trace data to create an intermediate representation of the execution. The underlying database can then be queried to build the actual message flow analysis; the intermediate representation greatly simplifies this. The database can also be queried for other analysis purposes.

5.4.1 Processing

As explained in Section 5.3, the ROS 2 architecture contains multiple separate abstraction layers. The information collected by `ros2_tracing` [154] for analysis purposes is therefore spread out over all these layers. This also provides internal information about ROS 2. For example, the duration of the message publication call can be broken down into `rclcpp`, `rcl`, and DDS time. Hence, `ros2_tracing` instruments all layers in order to collect all relevant information. Furthermore, to minimize the runtime impact, the `ros2_tracing` instrumentation is split into two distinct groups: initialization and runtime. The initialization instrumentation points collect one-time information, in order to minimize the size of the data collected by the runtime instrumentation points, which are executed more often. Therefore, we need to combine data from multiple instrumentation points in order to get the high-level information we need. Moreover, `ros2_tracing` does not include instrumentation for the chosen DDS implementation; thus we instrumented it and combined all this information.

For example, when a new publisher object is created, 3 tracepoints are triggered: the first one is in `rcl`, the second one is in `rmw`, and the last one is in the DDS implementation. By combining the execution information collected at different levels of the ROS 2 architecture by the 3 tracepoints into one publisher entry in the database, we can attribute the `rcl`-, `rmw`-, or DDS-level data, of the subsequent publication instance trace events, to the corresponding publisher. To correlate and merge the information from multiple trace events, unique identifiers are required. In most cases, `ros2_tracing` uses the values of the pointers to the underlying internal data structures, i.e., memory addresses. To combine DDS-level information with the above ROS 2 information, we use the globally-unique identifier (GUID or GID) of the DDS data writer, which is the DDS-level object that actually sends messages from a ROS 2 publisher. This GID is used internally in ROS 2 and is part of the `rmw` interface. We have instrumented both eProsima Fast DDS [79] and Eclipse Cyclone DDS [78]. As a result of the above design, either DDS implementation can be used without affecting the model.

Our intermediate representation needs to be valid when tracing multiple processes on one host computer, and when combining data from multiple hosts. Unfortunately, memory addresses are only valid for one process. To account for multiple processes on the same host, we combine the pointer value with the process ID (PID). Then, to account for multiple hosts, we combine the pointer value & PID with a unique host ID obtained from the trace data. This 3-tuple is thus unique across different processes and hosts.

In summary, the resulting intermediate representation database contains information about all ROS 2 objects: nodes, publishers, subscriptions, and timers. It also contains all relevant instances: message publication instances, and subscription & timer callback instances. This database can then be queried for analysis purposes.

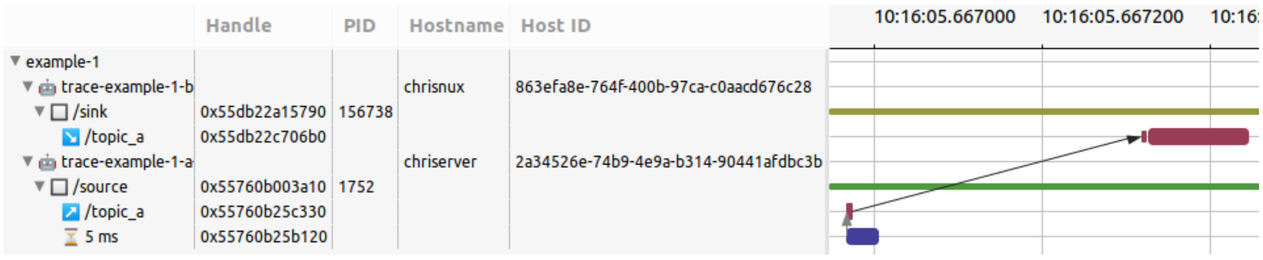
5.4.2 Implementation Details

To build a database of raw trace data as an intermediate representation, as described in the previous section, we used Eclipse Trace Compass [1], an open-source trace analysis framework. Since traces collected from multiple computers usually do not have the same clock reference, the traces need to be synchronized for the combined data to be valid, time-wise. Trace Compass can synchronize traces from distributed systems using network packet data collected from the kernel [51]. System clocks can also be synchronized directly using NTP [156]. The time synchronization method and its precision should of course be taken into consideration when extracting time-related information from the database. We then use Trace Compass to perform our message flow analysis using information from the intermediate execution representation database.

5.5 Message Flow Analysis

Our proposed message flow analysis builds a graph of the path of messages across a ROS 2 system using the information from the intermediate execution database, described in the previous section. However, to achieve this, we must add more information to the database and combine multiple elements. We must first track messages as they are sent over the network transport, to link a message being published by a publisher to the same message being received by one or more subscriptions. Then we add causal links, between messages that act as an input to a node, to the messages that are published by that node as the output. Finally, we put everything together to build the flow graph.

5.5.1 Transport Links



	Handle	PID	Hostname	Host ID
▼ example-1				
▼ trace-example-1-b			chrisnux	863efa8e-764f-400b-97ca-c0aacd676c28
▼ /sink	0x55db22a15790	156738		
/topic_a	0x55db22c706b0			
▼ trace-example-1-a			chriserver	2a34526e-74b9-4e9a-b314-90441afdbc3b
▼ /source	0x55760b003a10	1752		
/topic_a	0x55760b25c330			
5 ms	0x55760b25b120			

Figure 5.2 Transport link example. The tree structure on the left represents traces, with publishers, subscriptions, and timers under the nodes of each trace. To the right of this, internal handles, PIDs, and host information are shown: this is the 3-tuple needed to uniquely identify ROS 2 objects (see Section 5.4.1). Then, on the right is a time-based chart, which provides an abstract representation of the execution using time segments and arrows. In this example, a 5 ms timer triggers a callback which publishes a message on `/topic_a` under node `/source`. This message is received by the `/topic_a` subscription of node `/sink` on the other computer. Next to timers and subscriptions, segments represent the duration of a specific callback instance, from beginning to end. The smaller segment before the subscription callback segment represents the message being fetched (or *taken*) from the underlying middleware, before it is provided to the callback instance. For publishers, the segments represent the duration between the initial user-level publication call and the underlying DDS call. The longer arrow between the `/topic_a` publisher and subscription represents the transport link, i.e., the message going from the publisher to the subscription over the network. The shorter arrow between the 5 ms timer callback and the `/topic_a` publisher shows that the message was published during the timer callback.

Transport links associate a publication instance to the corresponding subscription callback instance on the other end. These links are always one-to-many, since messages always originate from a single publisher but can be received by any number of subscriptions.

ROS 2 internally provides metadata for all received messages, including the GID of the source publisher and the timestamp of the time right before DDS sent the message over the network. This information is collected on the subscription side using an instrumentation point in `rmw`. The publisher GID is collected during initialization; the source timestamp is collected on the publisher side using DDS instrumentation, since this information is not made available to ROS 2. To uniquely identify messages and thus avoid collisions in case multiple publishers emit messages at the same time, we should use a combination of the publisher GID and the source timestamp. Unfortunately, as of writing this, a bug in the implementation of the `rmw` interface for Cyclone DDS prevents us from relying on the GID. We therefore instead reduce the probability of collisions by combining the source timestamp with the topic name, which is known on both sides of the transport link. Furthermore, unless the source clock has a higher granularity, collisions are unlikely, given that source timestamps have nanosecond-level precision. These elements are all available from the intermediate execution representation database.

Collecting this low-level execution information, to track messages, allows our method to work transparently, i.e., without needing to modify a system to add fields to messages or rely on high-level tracking logic, unlike what is done by [143, 148]. More importantly, we expect the overhead to be much smaller, given the low overhead of the `ros2_tracing` instrumentation, as demonstrated in [154]. Fig. 5.2 shows an example of a transport link, with a subscription on one computer receiving a message from a publisher on another computer. Given our technique for tracking messages and uniquely identifying ROS 2 objects (see Section 5.4.1), there is no difference between a transport link between two computers, and a transport link constrained to a single computer.

5.5.2 Causal Message Links

For causal links, we define the causality of messages based on both time and value. In direct cases, an output message is generated and published when a new input message is received and processed, thus linking the two messages. In indirect cases, an input message is linked to an output message if the content of the former is used to generate the content of the latter, without any strict requirements on time. Indeed, the link is not strictly time-related, since causal links can be asynchronous, as we will explain in the following.

Direct Case

For the direct case, new messages are published on any number of topics directly during the subscription callback for a received message. The input message is thus linked to all messages that are published between the start and end of the corresponding subscription callback instance on the same thread. Since normal subscription callbacks only process a single message, the causal link for the direct case is strictly one-to-many. No user-level annotation is necessary for the direct case. An example is shown in Fig. 5.3, with a pipeline of three nodes and direct one-to-one causal links. In this case, the message flow graph generated from this exchange would be visually identical, since there are no additional links to be found.

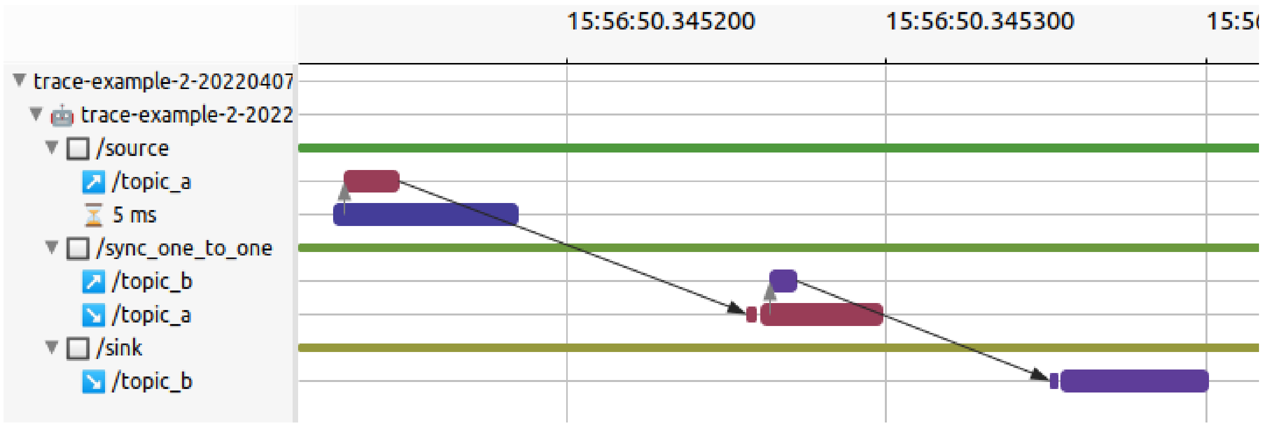
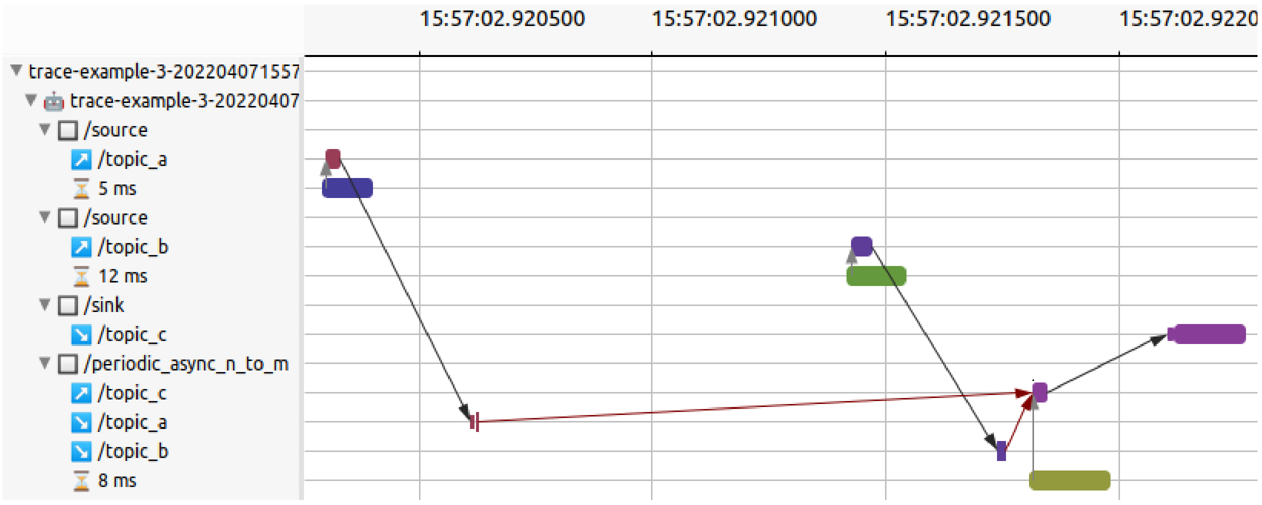


Figure 5.3 Direct causal message link example. A message is published on `/topic_a` during a 5 ms timer callback by node `/source`. The message is then received by the corresponding subscription under node `/sync_one_to_one`. During the subscription callback for that message, a message is published on `/topic_b`, which is finally received by the corresponding subscription under node `/sink`. The link between the input message and the output message is therefore a direct one-to-one causal link.

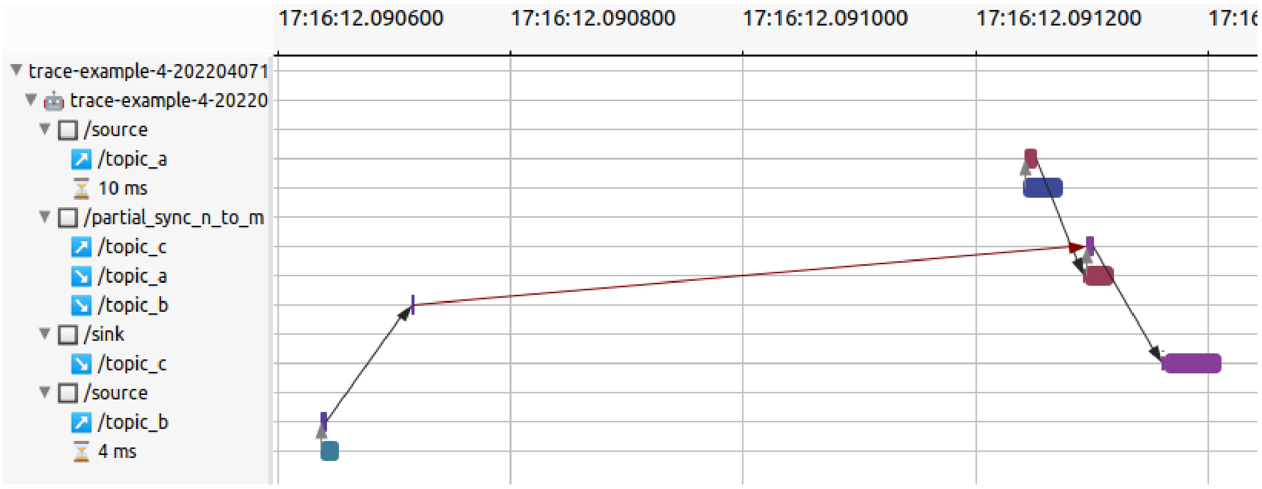
Indirect Case

For the indirect case, causal links are the result of user-level code, i.e., above the ROS 2 API. We therefore cannot detect these causal links from the trace itself; users need to provide this application-specific information for the links to be detected. We achieve this using simple annotations in the form of one-time tracepoints during the initialization phase, after the subscriptions and publishers have been created. Annotations simply contain the type of message link and the list of input subscriptions and output publishers. Annotation can thus be easily added to an existing system, without needing to modify the existing application logic or message structures as is done by [143, 148].

From studying real systems and the Autoware reference system [100], we define two types of causal links: periodic asynchronous link and partial synchronous link, both N-to-M, i.e., many-to-many. With the periodic asynchronous link, messages are received from N topics and are cached. A timer periodically triggers a callback during which the last message from each of the N caches is used to compute a result and then publish messages on M topics. For the partial synchronous link, messages are received from N topics and are cached as well. However, the result is conditionally computed during the subscription callback itself: if all N



(a) Periodic asynchronous causal message link. All messages received by node `/periodic_async_n_to_m` are cached. The periodic callback triggered by the 8 ms timer then uses those cached messages to compute and publish an output message on `/topic_c`. Therefore, the subscription callback of the input messages (`/topic_a` and `/topic_b`) are linked to the output message publication (`/topic_c`). These two periodic asynchronous links are shown in red.



(b) Partial synchronous causal message link. Messages are received by node `/partial_sync_n_to_m`. The first message (`/topic_b`) is received and cached, since the other cache is empty. However, when the second message (`/topic_a`) is received, both messages are available, and are therefore used to compute and publish an output message on `/topic_c`. Therefore, the subscription callback of the first message (`/topic_b`) is linked to the output message publication (`/topic_c`) which happens during the subscription callback for the second message (`/topic_a`).

Figure 5.4 Indirect causal message links examples: (a) periodic asynchronous and (b) partial synchronous. In both examples, messages are published periodically using timers on `/topic_a` and `/topic_b` by two `/source` nodes. These messages are then received by subscriptions under the `/periodic_async_n_to_m` and `/partial_sync_n_to_m` node, respectively. An output message linked to the input messages is eventually published on `/topic_c`.

caches contain a message, a result is computed and published on M topics. The caches are then reset so that at least one new message from each of the N input topics is received again before the next output. Without the link annotation, the partial synchronous link would be similar to the direct case; however, it would only link one out of N real input messages.

With the information from the annotations and the timer & subscription callback, and message publications instances from the intermediate execution representation database, we can thus automatically infer indirect causal links between specific input and output messages. Therefore, unlike [155], users do not need to provide these links after the fact, since they are already in the trace data. Furthermore, unlike the direct case, which is limited to a single input message per link, indirect causal links can be many-to-many, given their asynchronous nature. Fig. 5.4a shows an example for the periodic asynchronous link, while Fig. 5.4b shows an example for the partial synchronous link. In both cases, two input messages result in one output message. However, the mechanics of the two causal links are different. For the periodic asynchronous link, the output rate and delay between input and output depend on the period value for the timer. For the partial synchronous link, the output rate only depends on the rate of the inputs.

5.5.3 Building the Message Flow Graph

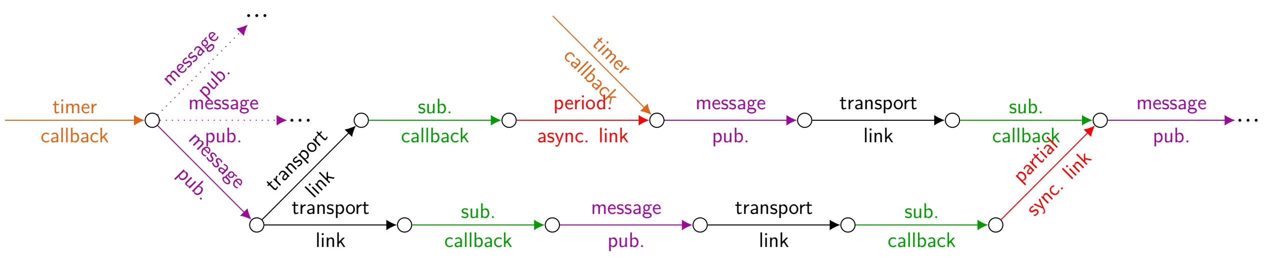


Figure 5.5 Simplified representation of a typical message flow graph, showing all edge types. Edges are segments of the message flow, and their duration is their weight. Vertices link one or more input edges to one or more output edges. The third message (bottom) has two outgoing transport links, i.e., it is received by two subscriptions. The first message (top) is processed by a subscription callback and put into a message cache. This message has a periodic asynchronous causal link to an output message generated and published by a timer callback (above). This last message is then received and processed by a subscription callback, which uses it along with another message linked by a partial synchronous causal link (below) to generate and publish a final message.

Information from the intermediate execution representation database, including transport links and direct causal message links, can be displayed directly to provide a visual representation, as shown in Fig. 5.2 and Fig. 5.3. We then need to use this information to build the message flow graph for a particular message, as selected by a user in the Trace Compass GUI.

As presented by Casini *et al.* [126] and used by [127–129], ROS computation graphs can be modeled as directed acyclic graphs (DAGs). The flow of a message can therefore also be modeled as a DAG. Fig. 5.5 shows a simplified version of a typical message flow graph. Message flow graphs have a limited set of edge types. Each edge type can be preceded and followed by a specific set of other edge types. For example, transport link edges are preceded by message publication edges and followed by subscription callback edges. Using this logic, and the different data sources presented in previous sections, the message flow graph can be constructed recursively, one edge at a time, going both forward and backward from the initial edge selected by the user.

Unlike [146], which assumed for simplification purposes that the message flow graph is a directed graph that only contains one-to-one links, our method supports one-to-many transport links and many-to-many causal message links. Furthermore, unlike [146] again, our method also builds the message flow graph both forward and backward from the initial element. This can be useful when analyzing traces from a ROS 2 system: building a message flow graph starting from one of the roots of the ROS 2 computation DAG will be different from a graph built starting from the leaf of a computation DAG, even if the resulting message flow graphs intersect. The initial segment from which to build the message flow graph can thus be chosen depending on the user needs.

Finally, there is one exception when modeling ROS computational graphs as DAGs. As explained by Blass *et al.* [128], the `/tf` topic is a special topic. It is used to communicate information about relationships between coordinate frames (*transforms*). All nodes that need and provide this information both subscribe and publish to the same topic, thus seemingly creating a loop in the graph model. However, `/tf` messages have a field which identifies the two coordinate frames to which the transform message applies. Therefore, a node publishing a `/tf` message might also receive that same message; however, it will not be used. As Blass *et al.* [128] do, we can detect the transport links for `/tf` with the same node as the source and destination, and remove them, since we assume that these messages are not meant for the originating node itself, and will not be used by it. However, this does not actually cause loops in our message flow graph implementation. Indeed, the subscriptions to the `/tf` topic are special subscriptions that are managed by ROS 2: messages are received and put into a buffer, the content of which is used eventually by the user. This application-level link is not detected; therefore, our method does not detect any message flow segments after subscription callbacks for `/tf` messages.

5.6 Experiments

We apply our proposed method first to a synthetic system and then to a real system. These experiments demonstrate how our technique can be used to analyze ROS 2 itself, as well as application-level logic, for performance optimization purposes. The code and instructions for these experiments are available in our repository.

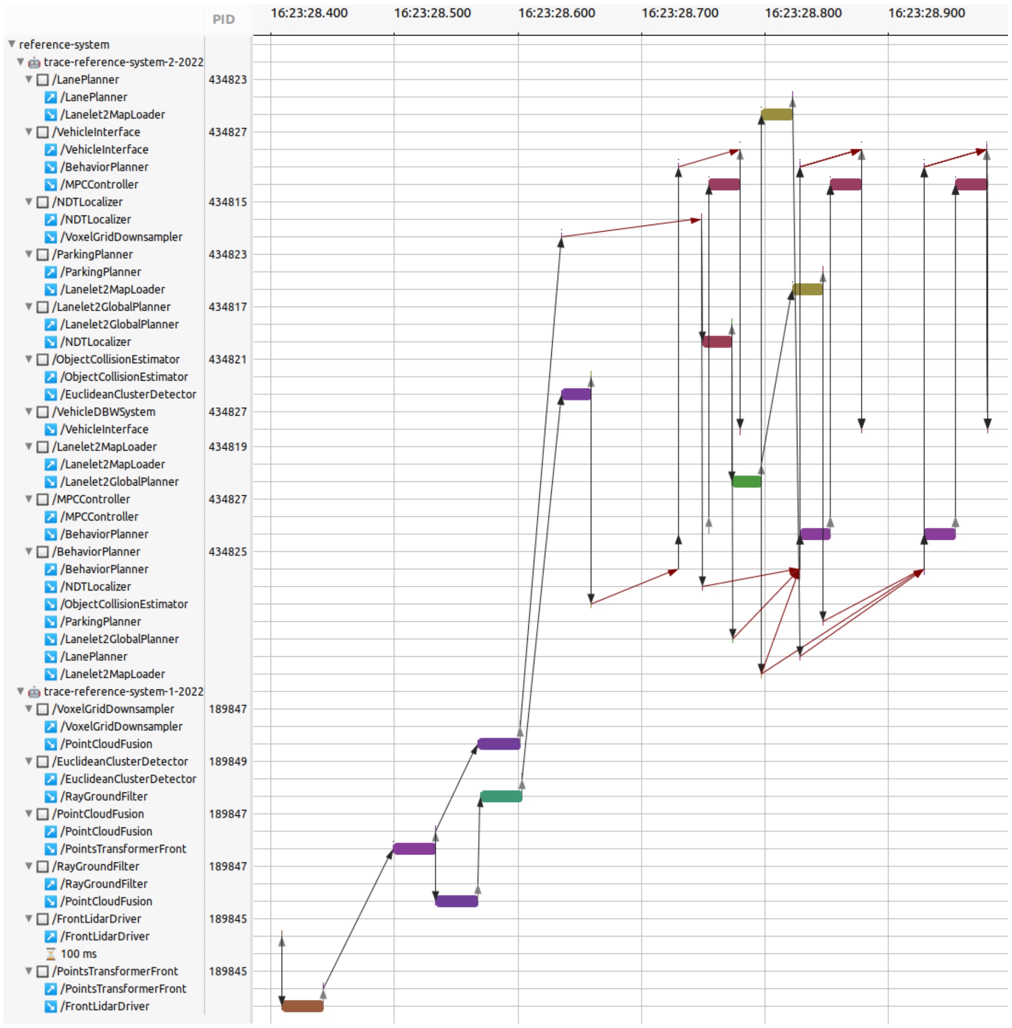
5.6.1 Autoware Reference System

We first apply our method to a reference system [100] with a synthetic computation graph based on Autoware [98,99], a ROS-based autonomous driving stack. The nodes and topics are based the Autoware system; however, all messages have the same type, and computation is replaced with a processing-intensive task, in order to consume CPU time. The computation graph has multiple inputs and outputs, i.e., sensor data and vehicle commands or secondary visualization outputs. It uses all types of causal message links, as defined in Section 5.5.2. However, it specifically uses the periodic asynchronous link in a many-to-one configuration, and the partial synchronous link in a two-to-one configuration.

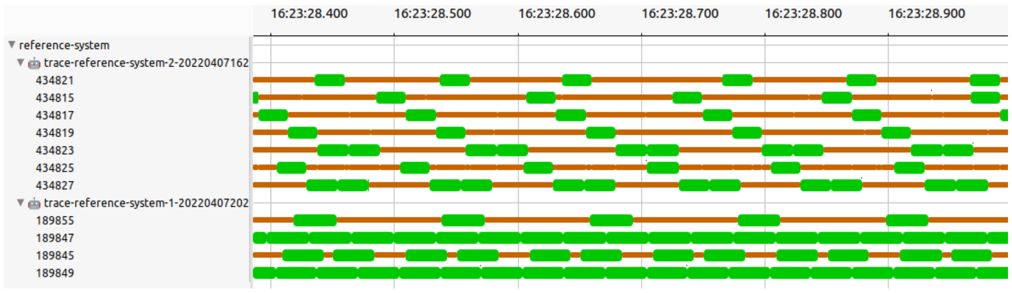
We split the nodes defined in the reference system into multiple executables and split those executables into two launch files. Each launch file is run on a specific host, with the two hosts being on the same network. We set the launch files to configure the LTTng tracer using `ros2_tracing` and enable all ROS 2 and DDS tracepoints. Furthermore, we enable network-related events, in order to synchronize the traces using [51].

Fig. 5.6 shows the entire message flow graph starting from one of the lidar drivers, which is one of the roots of the computation graph, along with the state of the executors over time. As displayed in Fig. 5.6a, the initial `/FrontLidarDriver` message results in three separate `/VehicleInterface` messages to the `/VehicleDBWSystem` node. This is due to the caching and asynchronous nature of the indirect causal links, which results in one-to-many or many-to-many links, as explained in Section 5.5.2. The end-to-end latency ranges from 370 ms for the first message to 571 ms for the third message.

Furthermore, as displayed in Fig. 5.6b, a secondary visualization shows the state of all executor instances over time. In this experiment, we only use the default single-threaded executor, which means that timer and subscription callbacks within a single process can only be processed one at a time. We can see that some executor instances are busier than others; if executors are too busy, there can be a greater delay between message re-

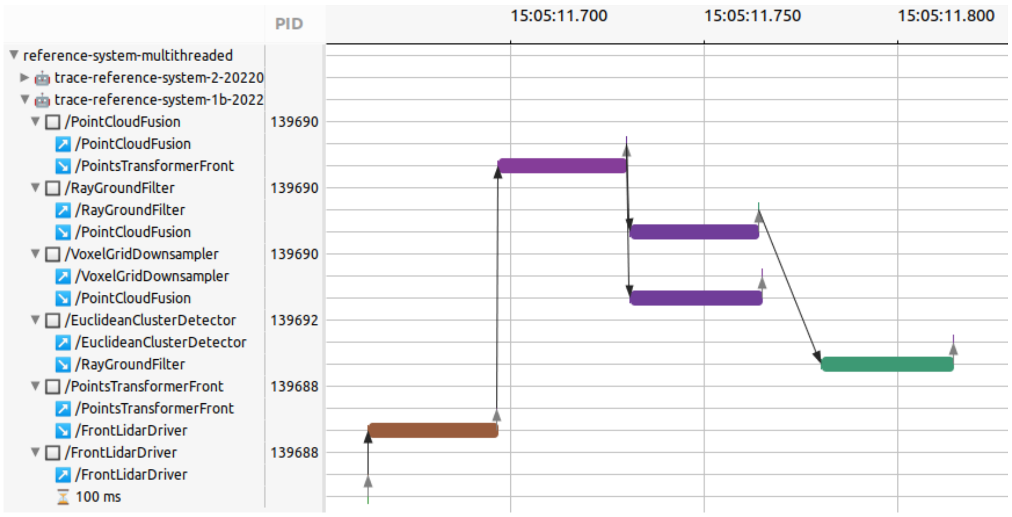


(a) End-to-end message flow graph. The initial root of the message flow graph is a 100 ms timer callback instance which publishes a `/FrontLidarDriver` message, while the main leaves of the graph are `/VehicleInterface` messages received by the `VehicleDBWSystem` node. The message flow graph includes direct and indirect causal links, including both periodic asynchronous and partial synchronous links.

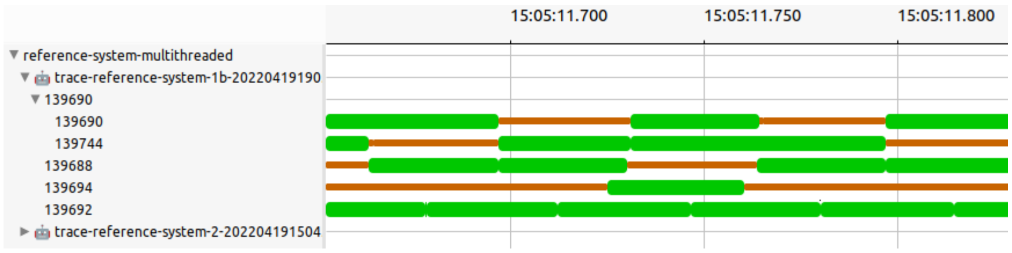


(b) State of all executor instances over time for the same time range as (a). For this executor state visualization, the tree structure on the left lists IDs of processes under each trace. On the right, the state of the executor instance for each process is displayed over time. Green segments represent execution instances (e.g., timer callback, subscription callback). Orange segments indicate the moments during which ROS 2 was waiting for new messages from the underlying middleware. Therefore, for a given process ID, orange means that the executor has nothing to process, while green means that the executor is busy executing callbacks. Red segments represent internal executor processing, although they are too small to be visible for this time range.

Figure 5.6 Autoware reference system (a) message flow result example and (b) executor state for the same time range.



(a) Partial message flow graph; segments for the second host are hidden. Callbacks for the same `/PointCloudFusion` message received by the `/RayGroundFilter` and `/VoxelGridDownsampler` nodes are executed concurrently.


15:05:11.700
15:05:11.750
15:05:11.800
reference-system-multithreaded
trace-reference-system-1b-20220419190
139690
139690
139744
139688
139694
139692
trace-reference-system-2-202204191504

(b) State of executor instances for the first host over time for the same time range as (a). Process 139690 has two executor threads (139690 and 139744), while other processes use single-threaded executors.

Figure 5.7 Autoware reference system (a) message flow and (b) executor state for the same time range, showing the impact of a multi-threaded executor instance.

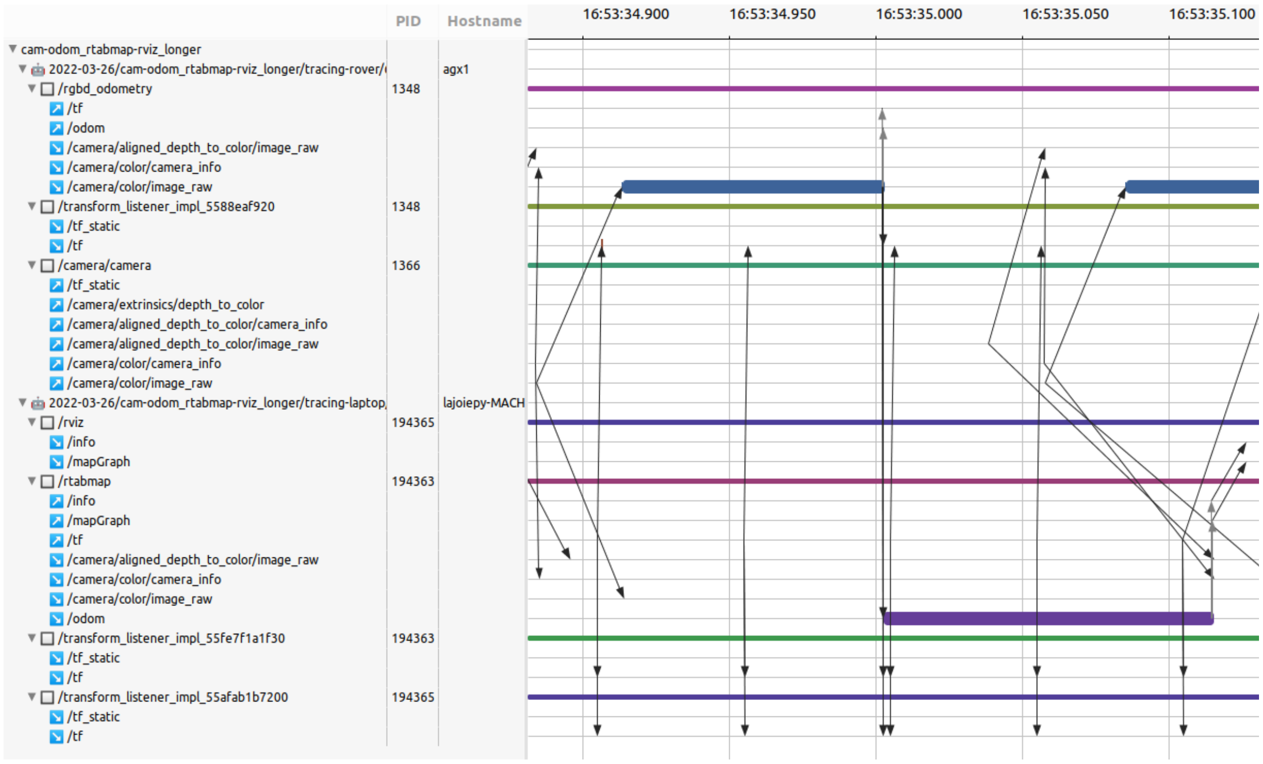
ception and processing. Depending on subscription options, old messages could be dropped, which wastes the CPU time used for publishing those dropped messages, thus resulting in a generally poor performance optimization. For example, unlike all executor instances under `trace-reference-system-2`, the executor instance for process 189847 is always busy, which could explain why the callback for the `/PointsTransformerFront` message under the `/PointCloudFusion` node happens long after the message was published. Also, the `/VoxelGridDownsampler`, `/PointCloudFusion`, and `/RayGroundFilter` nodes are all on the same process and thus share the same single-threaded executor instance. The callbacks for the same `/PointCloudFusion` message under two different nodes thus cannot be processed at the same time. This directly affects the end-to-end latency, as our method helps highlight.

In this case, the computation graph distribution could be improved: nodes could be split over more processes, and better executor designs could be used. For instance, as demonstrated in Fig. 5.7, to allow the callbacks of the `/VoxelGridDownsampler` and `/RayGroundFilter` nodes to run simultaneously, we can use a multi-threaded executor with 2 threads. As shown in Fig. 5.7b, there are two executor threads for the corresponding process (139690 and 139744). The subscription callbacks can then run concurrently, as shown in Fig. 5.7a. The end-to-end latency is hence reduced from 370 ms in the previous example to 298 ms in this example. Our method can therefore be used to compare or study the impact of proposed executor designs in order to address the open problems summarized in Section 5.2.

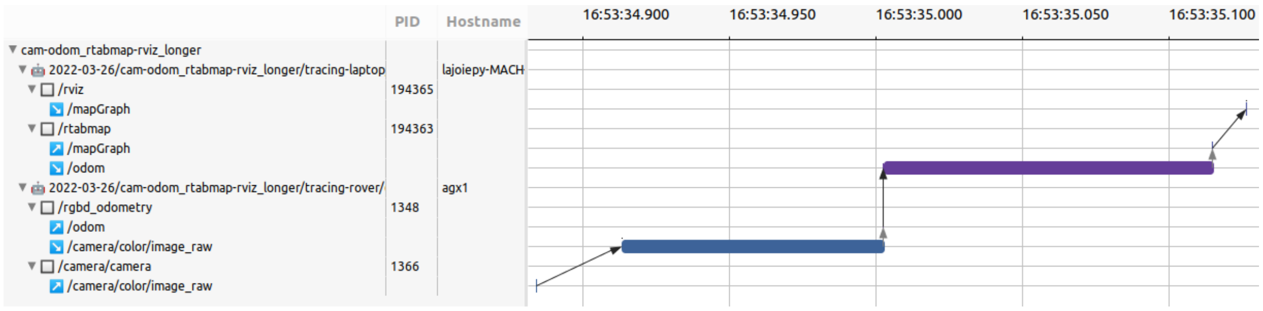
5.6.2 RTAB-Map

For our second experiment, we distribute the RTAB-Map [157] simultaneous localization and mapping (SLAM) system over two computers and trace it. One host runs the camera driver node and odometry node, while the other host runs the main SLAM node and rviz to visualize the resulting map. To obtain synchronized traces, we synchronize the clocks of the two hosts using NTP [156].

A section of the trace is represented in Fig. 5.8. Fig. 5.8a shows all callback instances, message publications, transport links, and direct links from the intermediate execution representation database for this time range. Fig. 5.8b shows an end-to-end message flow graph for the main computation pipeline. Using Trace Compass, we find that the end-to-end latency is 242.3 ms; the duration of the first subscription callback (odometry computation) is 89.5 ms, and the duration of the second subscription callback (RTAB-Map) is 112.5 ms. As seen shortly after the 16:53:35.000 time mark under the `/rgbd_odometry` node, a `/tf` message is published during a subscription callback instance. As mentioned in Section 5.5.3, the message is received by a special subscription, a transform listener, under the same process (PID 1348).


PID
Hostname
16:53:34.900
16:53:34.950
16:53:35.000
16:53:35.050
16:53:35.100
cam-odom_rtabmap-rviz_longer
2022-03-26/cam-odom_rtabmap-rviz_longer/tracing-rover/
agx1
/rgbd_odometry
1348
/tf
/odom
/camera/aligned_depth_to_color/image_raw
/camera/color/camera_info
/camera/color/image_raw
/transform_listener_impl_5588eaf920
1348
/tf_static
/tf
/camera/camera
1366
/tf_static
/camera/extrinsics/depth_to_color
/camera/aligned_depth_to_color/camera_info
/camera/aligned_depth_to_color/image_raw
/camera/color/camera_info
/camera/color/image_raw
2022-03-26/cam-odom_rtabmap-rviz_longer/tracing-laptop
lajoiepy-MACH
/rviz
194365
/info
/mapGraph
/rtabmap
194363
/info
/mapGraph
/tf
/camera/aligned_depth_to_color/image_raw
/camera/color/camera_info
/camera/color/image_raw
/odom
/transform_listener_impl_55fe7f1a1f30
194363
/tf_static
/tf
/transform_listener_impl_55afab1b7200
194365
/tf_static
/tf

(a) Display of intermediate execution representation data: subscription callbacks, message publications, transport links (black arrows), and direct causal links (gray arrows). A camera driver node (`/camera/camera`) and odometry node (`/rgbd_odometry`) run on the the first host (top half) along with a transform listener node (see Section 5.5.3). The RTAB-Map (`/rtabmap`) and rviz (`/rviz`) nodes run on the second host (bottom half) along with two transform listener nodes.



(b) End-to-end message flow graph for the main RTAB-Map computation pipeline, which goes through, in order: camera driver node (`/camera/camera`), odometry node (`/rgbd_odometry`), RTAB-Map node (`/rtabmap`), and rviz node (`/rviz`). The graph was generated starting from the subscription callback for a `/mapGraph` message received by the `/rviz` node, which is the last segment of the computation pipeline. Therefore, the message flow only goes backward from there to the root `/camera/color/image_raw` message published by the `/camera/camera` node on the other host. Looking at Fig. 5.8a, we know that generating the message flow graph starting from the initial `/camera/color/image_raw` message would result in a message flow graph with one-to-many links, similar to Fig. 5.6a.

Figure 5.8 RTAB-Map (a) callback instances and message publications along with transport and direct links, and (b) message flow result for the main computation pipeline for the same time range.

However, this `/tf` message is actually only intended for the two transform listeners on the other hosts (PIDs 194363 and 194365), and does not cause a loop since there are no further segments after these `/tf` messages are received. Our method could be improved to model and detect indirect causal links after `/tf` messages.

5.7 Runtime Overhead Evaluation

Since runtime overhead should be minimal to avoid perturbing an application [4], we also evaluate the overhead of execution data collection. In previous work, we demonstrated that `ros2_tracing` [154] introduces a mean end-to-end latency overhead of 0.0033 ms for a single message publication (i.e., publisher to subscription). Since the instrumentation proposed in [154] includes 10 tracepoints in the publish-subscribe hot path, this is comparable to a runtime cost per LTTng userspace tracepoint of 158 ns, as measured by [13]. Depending on the DDS implementation, our proposed method adds either 2 or 3 additional tracepoints to the hot path. However, the systems presented in Section 5.6 include between 3 and 8 message transport instances, and have end-to-end latencies ranging from 240 ms to 370 ms. Furthermore, as discussed in [154], the combination of a high-level ROS 2 scheduler on top of the OS scheduler and networking stack introduces a lot of variability. Therefore, we expect the end-to-end latency overhead for real applications to be small, especially when compared to the absolute latency.

We create a computation graph similar to the experiments in Section 5.6, with 5 one-to-one message transport instances and an expected end-to-end latency of approximately 260 ms. We use an Intel i7-3770 (3.40 GHz) 4-core CPU, 8 GB RAM system with Ubuntu 20.04.2, and disable power-saving features. We run the computation graph at 10 Hz for 20 minutes first without and then with tracing to compare the runtime impact of tracing on the end-to-end latency. By comparing the latencies for each case, shown in Fig. 5.9, we obtain a difference of means of 0.1597 ms and a difference of medians of 0.0521 ms. This end-to-end latency overhead is small compared to a total latency of 260 ms; we therefore consider it suitable for real applications. Furthermore, this value is within an order of magnitude of overhead values extrapolated from results by [154] and [13], respectively 0.0215 ms and 0.0103 ms. Finally, as mentioned previously, we expect this overhead to be less noticeable – and challenging to actually measure – on more complex ROS 2 systems.

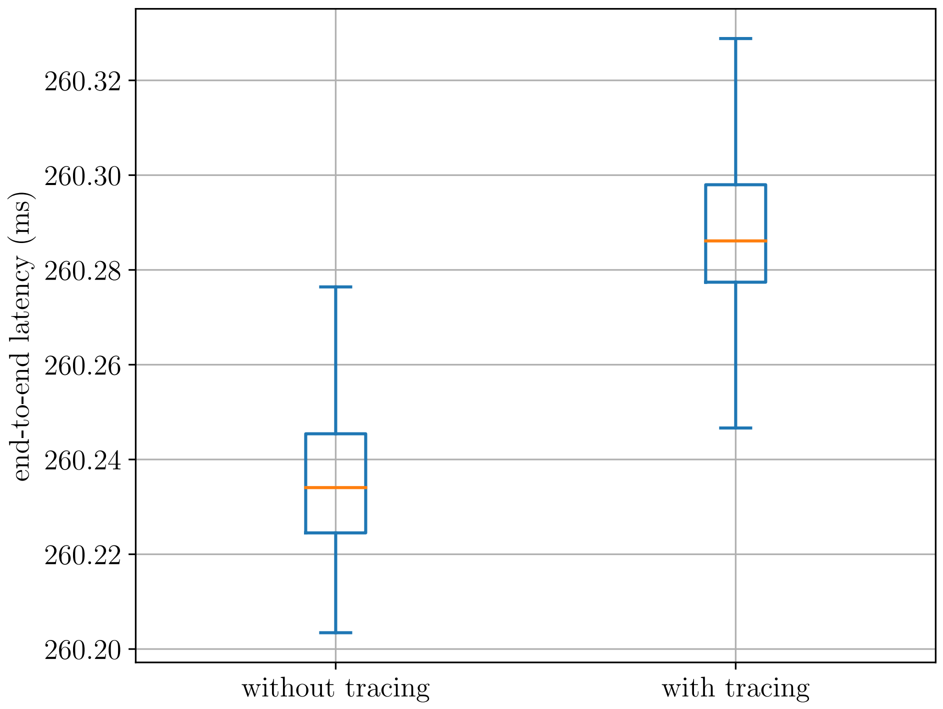


Figure 5.9 End-to-end latency comparison, without tracing (left) and with tracing (right).

5.8 Future Work

Many improvements and additions could be made to our proposed message flow analysis method and executor state visualization.

First, our method can easily be extended with other indirect causal links. Moreover, transport links could be split into actual network transport time and a time delay between message reception by DDS and processing by the executor. As mentioned in Section 5.6.1, it would help highlight delays in callback executions when the executor is busy processing other callbacks. This would require additional instrumentation in the underlying DDS middleware. Other executor types, such as the multi-threaded executor or other executors presented in previous work [132–135], could also be instrumented and supported for the executor state visualization. Additionally, as mentioned in Section 5.5.3 and shown in Section 5.6.2, special subscriptions like the transform listener could be supported to be able to detect valid message flow segments resulting from received `/tf` messages.

Furthermore, our message flow analysis method could be further extended into a critical path analysis [62]. Fundamentally, indirect causal links (shown in red in Fig. 5.4, Fig. 5.5, and Fig. 5.6a) are wait intervals. Indeed, as mentioned in Section 5.5.2, the duration of a periodic asynchronous link depends on the period of the timer, and the duration of a partial synchronous link depends on the other input messages. These wait segments could thus be recursively replaced with the actual cause of the wait, as is done by Giraldeau and Dagenais [66] using kernel-level wait primitives for wait dependencies across a distributed system.

Similarly, the message flow graph could be augmented with other information. For example, as mentioned in Section 5.4.1, the duration of a message publication call can be broken down into `rclcpp`, `rcl`, and DDS time. The required information is already collected using `ros2_tracing` and available in the intermediate execution representation database. Finally, other metrics, such as message publication or reception rate and executor usage over time, could be extracted from the database and displayed alongside our proposed visualizations.

5.9 Conclusion

In conclusion, modern robotic systems are built as distributed computation graphs, using publish-subscribe frameworks such as ROS 2. However, there are open problems with the higher-level scheduling of tasks performed by the ROS 2 executor, which can affect performance.

We presented a low-overhead method for extracting and visualizing the flow of a message across a distributed ROS 2 system. Our novel approach can detect exchanges of messages across distributed systems, and also introduces simple annotations for indirect causal message links. This is achieved without needing to modify the applications themselves. Combined with a visualization of the state of the executor instances over time, this is useful for optimizing both application layers and ROS 2 itself.

Finally, the underlying intermediate execution representation data can be leveraged for further analyses. Furthermore, the message flow graph can also be extended with more information, and can be expanded into a critical path analysis, by recursively resolving wait dependencies resulting from indirect causal links.

CHAPTER 6 GENERAL DISCUSSION

This chapter aims to discuss the work presented in Chapters 4 and 5, and to summarize its research contributions as well as its potential impact on the research community, first with the instrumentation and tracing tools, and then with the message flow analysis method.

6.1 Instrumentation and Tracing Tools

Chapter 4 introduces a framework for tracing ROS 2. This includes instrumentation to extract ROS 2-level execution information, which is not limited to one specific performance metric or analysis goal, unlike existing techniques. Its low runtime overhead was established, which is again better than existing methods, making it compatible with real-time and observability constraints, as introduced in Section 1.2.2. Appendix A shows the full results for the overhead experiment presented in Section 4.6. Moreover, tracing configuration tools were integrated into the powerful ROS 2 orchestration system, which is paramount for it to be used effectively by both researchers and actual end-users of ROS 2, as explained in Section 1.2.3. This is a significant contribution to the research community, since it can be used for other endeavours without needing to focus on runtime execution information collection. The instrumentation can also be extended to cover other use-cases.

Furthermore, while the LTTng tracer was chosen for the research work presented in this thesis, the intentional split of instrumentation points and tracepoints into two distinct elements facilitates the use of other tracers. Finally, the same approach could be applied to other robotics frameworks.

6.2 Trace Data Analysis Method

Chapter 5 presents a trace data analysis method to extract and visualize the path of a message across a distributed robotic system. This method brings numerous contributions. First, it introduces an abstract model of the execution of a ROS 2 application which is compatible with traces obtained from distributed systems. This contribution has a high impact potential for the research community, since other researchers can leverage it for other analysis methods. Then, the message flow analysis method extracts high-level information from the raw trace data. This allows researchers and users to focus on the right sections of a trace for general performance analysis, as introduced in Section 1.2.4. It also allows studying high-level task schedulers, as introduced in Section 1.2.1 and explained in Section 2.4.3.

Moreover, as described in Chapter 5, considering – and correctly identifying – indirect causal links between input and output messages is important for the method to be valid for real, complex systems, otherwise its applicability would be poor. The proposed simple user-level annotation can be further extended for other types of causal links.

Furthermore, as an extension of the low-overhead tracing framework presented in Chapter 4, the low runtime overhead of the message flow analysis method was demonstrated. Finally, experiments on both synthetic and real robotic systems demonstrated its potential for performance optimization, and in general for understanding the execution of a ROS 2 system.

CHAPTER 7 CONCLUSION

The work presented in this thesis aims to improve software debugging and performance analysis tools & techniques in robotics. To achieve this, using new tools for extracting execution information from ROS 2, a method for visualizing the path of messages across distributed robotic systems was proposed.

7.1 Summary of Works

In summary, the research objectives presented in Section 1.3 were achieved through the work presented in Chapters 4 and 5. First, `ros2_tracing`, introduced in Chapter 4, accomplished the first objectives: the low-overhead instrumentation allows extracting execution information from ROS 2, and the tracing tools allow configuring tracing for complex ROS 2 systems. Then, the message flow analysis method presented in Chapter 5 accomplished the remaining objectives. Indeed, it extracts high-level information through the path of messages across distributed ROS 2 systems. It also includes an abstract model of the execution of a ROS 2 application, and includes experiments on both simulated and real robotic systems to validate its contributions and usefulness.

7.2 Limitations

Notwithstanding its contributions, the research work presented in this thesis still has some limitations. The main limitation is its focus on offline analysis. Some of the existing methods presented in Section 2.4.3 do online monitoring, leveraging the same kind of instrumentation, but processing the execution information during runtime to adjust system configurations, for example. While online monitoring inherently has a higher runtime impact, since it does more than just collect execution information during runtime, it can still provide a lot of value. However, both `ros2_tracing` and the message flow analysis presented in this thesis were developed for offline processing. Indeed, `ros2_tracing` was created for collecting all information during runtime and then performing trace data analysis after execution, and would thus need to be adapted for monitoring use-cases. Furthermore, as mentioned in Section 2.1.2, flight recorder-style tracing using the LTTng snapshot mode can be useful in production. However, due to the `ros2_tracing` two-phase instrumentation design (Section 4.4.1), to process a chunk of trace data generated during the runtime phase, the trace data generated during the initialization phase of the system is required. Therefore, the LTTng snapshot mode cannot be used as-is, since the chunk of runtime trace data written to disk after triggering a snapshot would be incomplete; this would need to be addressed.

7.3 Future Research

Other than working towards addressing the limitations mentioned in the previous section, the clear follow-up to the research work presented in this thesis is the extension of the message flow analysis to resolve wait dependencies. Indeed, similar to [66], wait dependencies in the message flow graph caused by indirect causal links could be resolved to find the actual cause of the wait, resulting in a critical path graph. Moreover, augmenting the message flow graph with actual processing information – for example, for image processing performed during subscription callbacks – would be interesting. Finally, the work and methods presented in this thesis could be utilized to perform a more comprehensive study of the ROS 2 executors to work on addressing the open problems identified in Section 2.4.3.

REFERENCES

[1] "Eclipse trace compass." [Online]. Available: https://www.eclipse.org/tracecompass/

[2] A. P.-V. Nguyen, "Méthodes d'inspection automatique d'infrastructure par robot mobile," Master's thesis, Polytechnique Montréal, 2017.

[3] P.-Y. Lajoie, "Simultaneous localization and mapping systems robust to perceptual aliasing," Master's thesis, Polytechnique Montréal, 2019.

[4] B. Gregg, *Systems Performance: Enterprise and the Cloud*, 2nd ed. Pearson, 2020.

[5] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it?" *arXiv preprint arXiv:1701.00854*, 2017.

[6] W. Heisenberg, "Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik," *Zeitschrift für Physik*, vol. 43, no. 3-4, pp. 172–198, 1927, English translation in "Quantum theory and measurement" by Wheeler and Zurek.

[7] "Nyquist–Shannon sampling theorem." [Online]. Available: https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem

[8] "`perf(1)`." [Online]. Available: https://man7.org/linux/man-pages/man1/perf.1.html

[9] "`gprof(1)`." [Online]. Available: https://man7.org/linux/man-pages/man1/gprof.1.html

[10] B. Gregg, "The flame graph," *Communications of the ACM*, vol. 59, no. 6, pp. 48–57, 2016.

[11] "Flamegraph: Stack trace visualizer." [Online]. Available: https://github.com/brendangregg/FlameGraph

[12] "`instrument_function gcc` plugin." [Online]. Available: https://github.com/christophebedard/instrument-attribute-gcc-plugin

[13] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–33, 2018.

[14] B. Gregg and J. Mauro, *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD.* Prentice Hall Professional, 2011.

[15] "Debugging the kernel using ftrace - part 1," 2009. [Online]. Available: https://lwn.net/Articles/365835

[16] S. Rostedt, "Finding origins of latencies using ftrace," *Proc. RT Linux WS*, 2009.

[17] T. Bird, "Measuring function duration with ftrace," in *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2009.

[18] "ftrace." [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[19] M. Kerrisk, *The Linux programming interface: a Linux and UNIX system programming handbook.* No Starch Press, 2010.

[20] M. K. Johnson and E. W. Troan, *Linux application development.* Addison-Wesley Longman Publishing Co., Inc., 1998.

[21] "`strace(1)`." [Online]. Available: https://man7.org/linux/man-pages/man1/strace.1.html

[22] A. Knüpfer *et al.*, "The vampir performance analysis tool-set," in *Tools for high performance computing.* Springer, 2008, pp. 139–155.

[23] V. Prasad *et al.*, "Locating system problems using dynamic instrumentation," in *2005 Ottawa Linux Symposium.* Citeseer, 2005, pp. 49–64.

[24] F. C. Eigler *et al.*, "Architecture of systemtap: a linux trace/probe tool," 2005.

[25] F. C. Eigler and R. Hat, "Problem solving with systemtap," in *Proc. of the Ottawa Linux Symposium.* Citeseer, 2006, pp. 261–268.

[26] B. Gregg, *BPF Performance Tools.* Addison-Wesley Professional, 2019.

[27] "ebpf." [Online]. Available: https://ebpf.io/

[28] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[29] ——, "Lttng, filling the gap between kernel instrumentation and a widely usable kernel tracer," 2009.

[30] M. Desnoyers, "Low-impact operating system tracing," Ph.D. dissertation, École Polytechnique de Montréal, 2009.

[31] W. Gropp *et al.*, *Using MPI: portable parallel programming with the message-passing interface.* MIT press, 1999, vol. 1.

[32] M. Desnoyers and M. Dagenais, "Low disturbance embedded system tracing with linux trace toolkit next generation," in *ELC (Embedded Linux Conference)*, vol. 2006. Citeseer, 2006.

[33] M. Desnoyers and M. R. Dagenais, "Lockless multi-core high-throughput buffering scheme for kernel tracing," *ACM SIGOPS Operating Systems Review*, vol. 46, no. 3, pp. 65–81, 2012.

[34] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, Las Vegas, NV, October 1998, pp. 509–518.

[35] M. Desnoyers and M. R. Dagenais, "Synchronization for fast and reentrant operating system kernel tracing," *Software: Practice and Experience*, vol. 40, no. 12, pp. 1053–1072, 2010.

[36] "Common trace format." [Online]. Available: https://diamon.org/ctf/

[37] M. Desnoyers and M. Dagenais, "Lttng: Tracing across execution layers, from the hypervisor to user-space," in *Linux symposium*, vol. 101, 2008.

[38] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais, "Combined tracing of the kernel and applications with lttng," in *Proceedings of the 2009 linux symposium.* Citeseer, 2009, pp. 87–93.

[39] "Babeltrace." [Online]. Available: https://babeltrace.org/

[40] "Kernelshark." [Online]. Available: https://kernelshark.org/

[41] "`trace-cmd(1)`." [Online]. Available: https://man7.org/linux/man-pages/man1/trace-cmd.1.html

[42] M. S. Müller *et al.*, "Developing scalable applications with vampir, vampirserver and vampirtrace." in *PARCO*, vol. 15, 2007, pp. 637–644.

[43] R. Schöne *et al.*, "The vampirtrace plugin counter interface: introduction and examples," in *European Conference on Parallel Processing.* Springer, 2010, pp. 501–511.

[44] "Vampir." [Online]. Available: https://vampir.eu/

[45] "Tracealyzer." [Online]. Available: https://percepio.com/tracealyzer/

[46] "Specification of diagnostic log and trace (4.3.1)." [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_DiagnosticLogAndTrace.pdf

[47] "Eclipse theia trace viewer extension." [Online]. Available: https://github.com/theia-ide/theia-trace-extension

[48] M. Woodside, S. Tjandra, and G. Seyoum, "Issues arising in using kernel traces to make a performance model," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 11–15.

[49] A. Duda *et al.*, "Estimating global time in distributed systems." in *ICDCS*, vol. 87, 1987, pp. 299–306.

[50] B. Poirier, R. Roy, and M. Dagenais, "Accurate offline synchronization of distributed traces using kernel-level events," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 75–87, 2010.

[51] M. Jabbarifar and M. Dagenais, "Liana: Live incremental time synchronization of traces for distributed systems analysis," *Journal of network and computer applications*, vol. 45, pp. 203–214, 2014.

[52] T. Bertauld and M. R. Dagenais, "Low-level trace correlation on heterogeneous embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2017, no. 1, pp. 1–14, 2017.

[53] R. Beamonte, "Traçage de systèmes linux multi-coeurs en temps réel," Master's thesis, École Polytechnique de Montréal, 2013.

[54] ——, "Runtime verification of real-time applications using trace data and model requirements," Ph.D. dissertation, École Polytechnique de Montréal, 2016.

[55] M. Côté and M. R. Dagenais, "Problem detection in real-time systems by trace analysis," *Advances in Computer Engineering*, vol. 2016, 2016.

[56] F. Rajotte and M. R. Dagenais, "Real-time linux analysis using low-impact tracer," *Advances in Computer Engineering*, vol. 2014, 2014.

[57] R. Beamonte and M. R. Dagenais, “Linux low-latency tracing for multicore hard real-time systems,” *Advances in Computer Engineering*, vol. 2015, 2015.

[58] L. Gelle, N. Ezzati-Jivan, and M. R. Dagenais, “Combining distributed and kernel tracing for performance analysis of cloud applications,” *Electronics*, vol. 10, no. 21, p. 2610, 2021.

[59] J. E. Kelley Jr and M. R. Walker, “Critical-path planning and scheduling,” in *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, 1959, pp. 160–173.

[60] M. Abramovici, P. R. Menon, and D. T. Miller, “Critical path tracing-an alternative to fault simulation,” in *20th Design Automation Conference Proceedings.* IEEE, 1983, pp. 214–220.

[61] N. Deo, *Graph theory with applications to engineering and computer science.* Courier Dover Publications, 2017.

[62] C.-Q. Yang and B. P. Miller, “Critical path analysis for the execution of parallel and distributed programs,” in *The 8th International Conference on Distributed.* IEEE Computer Society, 1988, pp. 366–367.

[63] J. K. Hollingsworth, “An online computation of critical path profiling,” in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1996, pp. 11–20.

[64] A. G. Saidi *et al.*, “Full-system critical path analysis,” in *ISPASS 2008-IEEE International Symposium on Performance Analysis of Systems and software.* IEEE, 2008, pp. 63–74.

[65] P.-M. Fournier and M. R. Dagenais, “Analyzing blocking to debug performance problems on multi-core systems,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 77–87, 2010.

[66] F. Giraldeau and M. Dagenais, “Wait analysis of distributed systems using kernel tracing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2450–2461, 2015.

[67] F. Doray and M. Dagenais, “Diagnosing performance variations by comparing multi-level execution traces,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 462–474, 2016.

[68] H. Nemati *et al.*, "Critical path analysis through hierarchical distributed virtualized environments using host kernel tracing," *IEEE Transactions on Cloud Computing*, 2019.

[69] N. Ezzati-Jivan *et al.*, "Depgraph: Localizing performance bottlenecks in multi-core applications using waiting dependency graphs and software tracing," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM).* IEEE, 2020, pp. 149–159.

[70] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s – a publish/subscribe protocol for wireless sensor networks," in *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08).* IEEE, 2008, pp. 791–798.

[71] "Mqtt." [Online]. Available: https://mqtt.org/

[72] A. S. Huang, E. Olson, and D. C. Moore, "Lcm: Lightweight communications and marshalling," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems.* IEEE, 2010, pp. 4057–4062.

[73] "Lcm." [Online]. Available: https://github.com/lcm-proj/lcm

[74] L. Völker, "Some/ip–die middleware für ethernet-basierte kommunikation," *Hanser automotive networks*, 2013.

[75] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 2003, pp. 200–206.

[76] "Dds." [Online]. Available: https://www.dds-foundation.org/

[77] S. Vinoski, "Corba: Integrating diverse applications within distributed heterogeneous environments," *IEEE Communications magazine*, vol. 35, no. 2, pp. 46–55, 1997.

[78] "Eclipse cyclone dds." [Online]. Available: https://github.com/eclipse-cyclonedds/cyclonedds

[79] eProsima, "Fast dds." [Online]. Available: https://github.com/eProsima/Fast-DDS

[80] "Rti connext dds." [Online]. Available: https://www.rti.com/products

[81] B. Gerkey *et al.*, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th international conference on advanced robotics*, vol. 1. Citeseer, 2003, pp. 317–323.

[82] M. Quigley *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[83] S. Kolak *et al.*, "It takes a village to build a robot: An empirical study of the ros ecosystem," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2020, pp. 430–440.

[84] S. Macenski *et al.*, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[85] "Ros 2." [Online]. Available: https://docs.ros.org/en/rolling/

[86] B. Gerkey, "Why ros 2?" [Online]. Available: https://design.ros2.org/articles/why_ros2.html

[87] D. Thomas, "Changes between ros 1 and ros 2." [Online]. Available: https://design.ros2.org/articles/changes.html

[88] ——, "Ros 2 middleware interface." [Online]. Available: https://design.ros2.org/articles/ros_middleware_interface.html

[89] "Eclipse iceoryx™ - true zero-copy inter-process-communication." [Online]. Available: https://github.com/eclipse-iceoryx/iceoryx

[90] "rmw_iceoryx." [Online]. Available: https://github.com/ros2/rmw_iceoryx

[91] C. Bédard, "Ros 2 over email: rmw_email, an actual working rmw implementation." [Online]. Available: https://christophebedard.com/ros-2-over-email/

[92] "rmw_email." [Online]. Available: https://github.com/christophebedard/rmw_email

[93] G. Biggs and T. Foote, "Managed nodes." [Online]. Available: https://design.ros2.org/articles/node_lifecycle.html

[94] J. Kay, "Proposal for implementation of real-time systems in ros 2." [Online]. Available: https://design.ros2.org/articles/realtime_proposal.html

[95] W. Woodall, "Ros 2 launch system." [Online]. Available: https://design.ros2.org/articles/roslaunch.html

[96] V. Mayoral-Vilches *et al.*, "Sros2: Usable cyber security tools for ros 2."

[97] "Executors." [Online]. Available: https://docs.ros.org/en/rolling/Concepts/About-Executors.html

[98] S. Kato *et al.*, “Autoware on board: Enabling autonomous vehicles with embedded systems,” in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 287–296.

[99] ——, “An open approach to autonomous vehicles,” *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[100] ROS 2 Real-Time Working Group, “Reference system.” [Online]. Available: https://github.com/ros-realtime/reference-system

[101] M. Reke *et al.*, “A self-driving car architecture in ros2,” in *2020 International SAUPEC/RobMech/PRASA Conference*. IEEE, 2020, pp. 1–6.

[102] NASA, “Viper’s mission operations.” [Online]. Available: https://www.nasa.gov/viper/lunar-operations

[103] J. Perron, “Viper: Volatiles investigating polar exploration rover,” in *ROS World 2021*. Open Robotics, October 2021. [Online]. Available: https://vimeo.com/649657650

[104] C. S. V. Gutiérrez *et al.*, “Real-time linux communications: an evaluation of the linux communication stack for real-time robotic applications,” *arXiv preprint arXiv:1808.10821*, 2018.

[105] ——, “Time-sensitive networking for robotics,” *arXiv preprint arXiv:1804.07643*, 2018.

[106] F. Reghenzani, G. Massari, and W. Fornaciari, “The real-time linux kernel: A survey on preempt_rt,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–36, 2019.

[107] Apex.AI, “performance_test.” [Online]. Available: https://gitlab.com/ApexAI/performance_test

[108] iRobot, “irobot ros 2 performance evaluation framework.” [Online]. Available: https://github.com/irobot-ros/ros2-performance

[109] Y. Maruyama, S. Kato, and T. Azumi, “Exploring the performance of ros2,” in *Proceedings of the 13th International Conference on Embedded Software*, 2016, pp. 1–10.

[110] C. S. V. Gutiérrez *et al.*, “Towards a distributed and real-time framework for robots: Evaluation of ros 2.0 communications for real-time robotic applications,” *arXiv preprint arXiv:1809.02595*, 2018.

[111] L. Puck *et al.*, "Distributed and synchronized setup towards real-time robotic control using ros2 on linux," in *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE).* IEEE, 2020, pp. 1287–1293.

[112] J. Kim *et al.*, "Security and performance considerations in ros 2: A balancing act," *arXiv preprint arXiv:1809.09566*, 2018.

[113] J. Fernandez *et al.*, "Performance study of the robot operating system 2 with qos and cyber security settings," in *2020 IEEE International Systems Conference (SysCon).* IEEE, 2020, pp. 1–6.

[114] P. Thulasiraman *et al.*, "Evaluation of the robot operating system 2 in lossy unmanned networks," in *2020 IEEE International Systems Conference (SysCon).* IEEE, 2020, pp. 1–8.

[115] S. Barut *et al.*, "Benchmarking real-time capabilities of ros 2 and orocos for robotics applications," in *2021 IEEE International Conference on Robotics and Automation (ICRA).* IEEE, 2021, pp. 708–714.

[116] H. Bruyninckx, "Open robot control software: the orocos project," in *Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164)*, vol. 3. IEEE, 2001, pp. 2523–2528.

[117] Y.-P. Wang *et al.*, "Tzc: Efficient inter-process communication for robotics middleware with partial serialization," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE, 2019, pp. 7805–7812.

[118] Z. Jiang *et al.*, "Message passing optimization in robot operating system," *International Journal of Parallel Programming*, vol. 48, no. 1, pp. 119–136, 2020.

[119] T. Kronauer *et al.*, "Latency analysis of ros2 multi-node systems," in *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI).* IEEE, 2021, pp. 1–7.

[120] L. Puck *et al.*, "Performance evaluation of real-time ros2 robotic control in a time-synchronized distributed network," in *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE).* IEEE, 2021, pp. 1670–1676.

[121] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[122] G. Fohler and K. Ramamritham, “Static scheduling of pipelined periodic tasks in distributed real-time systems,” in *Proceedings Ninth Euromicro Workshop on Real Time Systems.* IEEE, 1997, pp. 128–135.

[123] J. Fonseca *et al.*, “Response time analysis of sporadic dag tasks under partitioned scheduling,” in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES).* IEEE, 2016, pp. 1–10.

[124] Y. Cho *et al.*, “Conditionally optimal parallelization of real-time dag tasks for global edf,” in *2021 IEEE Real-Time Systems Symposium (RTSS).* IEEE, 2021, pp. 188–200.

[125] J. Peeck, J. Schlatow, and R. Ernst, “Online latency monitoring of time-sensitive event chains in safety-critical applications,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 2021, pp. 539–542.

[126] D. Casini *et al.*, “Response-time analysis of ros 2 processing chains under reservation-based scheduling,” in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[127] Y. Tang *et al.*, “Response time analysis and priority assignment of processing chains on ros2 executors,” in *2020 IEEE Real-Time Systems Symposium (RTSS).* IEEE, 2020, pp. 231–243.

[128] T. Blass *et al.*, “Automatic latency management for ros 2: Benefits, challenges, and open problems,” in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 2021, pp. 264–277.

[129] T. Blaß *et al.*, “A ros 2 response-time analysis exploiting starvation freedom and execution-time variance,” in *2021 IEEE Real-Time Systems Symposium (RTSS).* IEEE, 2021, pp. 41–53.

[130] C. Lienen and M. Platzner, “Reconros executor: Event-driven programming of fpga-accelerated ros 2 applications,” *arXiv preprint arXiv:2201.07454*, 2022.

[131] Y. Yang and T. Azumi, “Exploring real-time executor on ros 2,” in *2020 IEEE International Conference on Embedded Software and Systems (ICESS).* IEEE, 2020, pp. 1–8.

[132] R. Lange, “Callback-group-level executor for ros 2,” in *ROSCon Madrid 2018.* Open Robotics, September 2018. [Online]. Available: https://vimeo.com/292707644

[133] H. Choi, Y. Xiang, and H. Kim, "Picas: New design of priority-driven chain-aware scheduling for ros2," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 251–263.

[134] J. Staschulat, I. Lütkebohle, and R. Lange, "The rclc executor: Domain-specific deterministic scheduling mechanisms for ros applications on microcontrollers: work-in-progress," in *2020 International Conference on Embedded Software (EMSOFT)*. IEEE, 2020, pp. 18–19.

[135] J. Staschulat, R. Lange, and D. N. Dasari, "Budget-based real-time executor for micro-ros," *arXiv preprint arXiv:2105.05590*, 2021.

[136] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *International Workshop on Embedded Software*. Springer, 2001, pp. 166–184.

[137] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[138] I. Malavolta *et al.*, "Mining guidelines for architecting robotics software," *Journal of Systems and Software*, vol. 178, p. 110969, 2021.

[139] A. Afzal *et al.*, "A study on challenges of testing robotic systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 96–107.

[140] M. Quigley, B. Gerkey, and W. D. Smart, *Programming robots with ROS*. O'Reilly, 2015, ch. Debugging Robot Behavior.

[141] D. Forouher, J. Hartmann, and E. Maehle, "Data flow analysis in ros," in *ISR/Robotik 2014; 41st International Symposium on Robotics*. VDE, 2014, pp. 1–6.

[142] U. A. Acar *et al.*, "A core calculus for provenance," *Journal of Computer Security*, vol. 21, no. 6, pp. 919–969, 2013.

[143] T. Witte and M. Tichy, "Inferred interactive controls through provenance tracking of ros message data," in *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*. IEEE, 2021, pp. 67–74.

[144] I. Lütkebohle, "Determinism in ros – or when things break /sometimes/ and how to fix it...," in *ROSCon Vancouver 2017*. Open Robotics, September 2017. [Online]. Available: https://doi.org/10.36288/ROSCon2017-900789

[145] Bosch Corporate Research, "Ros 1 tracetools." [Online]. Available: https://github.com/boschresearch/ros1_tracetools

[146] C. Bédard, "Message flow analysis for ros through tracing," 2019. [Online]. Available: https://christophebedard.com/ros-tracing-message-flow/

[147] S. Rivera *et al.*, "Ros-fm: Fast monitoring for the robotic operating system (ros)," in *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS).* IEEE, 2020, pp. 187–196.

[148] K. Nishimura *et al.*, "Raplet: Demystifying publish/subscribe latency for ros applications," in *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).* IEEE, 2021, pp. 41–50.

[149] B. Gregg, "Linux performance." [Online]. Available: https://www.brendangregg.com/linuxperf.html

[150] "tracetools_analysis." [Online]. Available: https://gitlab.com/ros-tracing/tracetools_analysis

[151] P.-Y. Lajoie *et al.*, "Towards collaborative simultaneous localization and mapping: a survey of the current research landscape." [Online]. Available: http://arxiv.org/abs/2108.08325

[152] R. K. Dewangan, A. Shukla, and W. W. Godfrey, "Survey on prioritized multi robot path planning," in *2017 IEEE International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM)*, 2017, pp. 423–428.

[153] Z. Yan, N. Jouandeau, and A. A. Cherif, "A survey and analysis of multi-robot coordination," *International Journal of Advanced Robotic Systems*, vol. 10, no. 12, p. 399, 2013. [Online]. Available: https://doi.org/10.5772/57313

[154] C. Bédard, I. Lütkebohle, and M. Dagenais, "ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ros 2," *arXiv preprint arXiv:2201.00393*, 2022.

[155] Z. Li, A. Hasegawa, and T. Azumi, "Autoware_perf: A tracing and performance analysis framework for ros 2 applications," *Journal of Systems Architecture*, vol. 123, p. 102341, 2022.

[156] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[157] M. Labbé and F. Michaud, “Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation,” *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446, 2019.

APPENDIX A ROS2_TRACING RUNTIME OVERHEAD

Table A.1 presents individual data points for the results presented in Fig. 4.6, showing that the difference between the minimum and maximum values is not significant when tracing is disabled or enabled.

Table A.1 Comparison of Message Latencies with Minimum and Maximum Values

Message size (KiB)	**Rate (Hz)**	**Tracing (N/Y)**	**Min. (ms)**	**Avg. (ms)**	**Max. (ms)**	**Std. (ms)**
1	100	N	0.029	0.036	0.141	0.003
		Y	0.032	0.041	0.104	0.003
	500	N	0.029	0.035	0.085	0.003
		Y	0.032	0.039	0.086	0.004
	1000	N	0.029	0.035	0.072	0.003
		Y	0.031	0.038	0.072	0.003
	2000	N	0.028	0.036	0.082	0.003
		Y	0.031	0.039	0.085	0.003
32	100	N	0.043	0.049	0.092	0.002
		Y	0.045	0.053	0.205	0.004
	500	N	0.043	0.049	0.091	0.003
		Y	0.045	0.052	0.144	0.003
	1000	N	0.042	0.048	0.101	0.003
		Y	0.044	0.051	0.103	0.003
	2000	N	0.041	0.048	0.100	0.003
		Y	0.044	0.051	0.223	0.003
64	100	N	0.058	0.064	0.111	0.003
		Y	0.062	0.068	0.237	0.003
	500	N	0.057	0.063	0.114	0.003
		Y	0.060	0.066	0.132	0.003
	1000	N	0.056	0.063	0.123	0.003
		Y	0.059	0.066	0.148	0.003
	2000	N	0.055	0.063	0.122	0.003
		Y	0.058	0.066	0.114	0.003
256	100	N	0.131	0.136	0.272	0.003
		Y	0.132	0.138	0.174	0.004
	500	N	0.128	0.131	0.172	0.003
		Y	0.130	0.133	0.188	0.002
	1000	N	0.128	0.133	0.211	0.005
		Y	0.131	0.137	0.233	0.004
	2000	N	0.128	0.133	0.211	0.004
		Y	0.131	0.138	0.216	0.005